Management of the Internet and Complex Services

European Sixth Framework Network of Excellence FP6-2004-IST-026854-NoE

Deliverable D9.5
Autonomic Management: Challenges and Solutions

The EMANICS Consortium

Caisse des Dépôts et Consignations, CDC, France
Institut National de Recherche en Informatique et Automatique, INRIA, France
University of Twente, UT, The Netherlands
Imperial College, IC, UK
International University Bremen, IUB, Germany
KTH Royal Institute of Technology, KTH, Sweden
Oslo University College, HIO, Norway
Universidat Politecnica de Catalunya, UPC, Spain
University of Federal Armed Forces Munich, UniBwM, Germany
Poznan Supercomputing and Networking Center, PSNC, Poland
University of Zürich, UniZH, Switzerland
Ludwig-Maximilian University Munich, LMU, Germany
University College London, UCL, UK
University of Pitesti, UniP, Romania

For more information on this document or the EMANICS Project, please contact:

Dr. Olivier Festor
Technopole de Nancy-Brabois — Campus scientifique
615, rue de Jardin Botanique — B.P. 101
F—54600 Villers Les Nancy Cedex
France

Phone: +33 383 59 30 66
Fax: +33 383 41 30 79
E-mail: <olivier.festor@loria.fr>

Document Control

Title: Autonomic Management: Challenges and Solutions

Type: Public

Editors: Marinos Charalambides, George Pavlou

E-mail: m.charalambides@ee.ucl.ac.uk, g.pavlou@ee.ucl.ac.uk
Authors: Antonio Astorga, Themistoklis Bourdenas, Wei Koong Chai, Marinos Charalambides, Richard Clegg, Tiago Fioreze, David Hausheer, Fabio Hecht, Rick Hofstede, Raul Landa, Feng Liu, Javier Rubio Loyola, Emil Lupu, Cristian Morariu, Giovane Moura, George Pavlou, Aiko Pras, Peter Racz, Miguel Rio, Joan Serrat, Anna Sperotto, Bjorn Stelte, Burkhard Stiller, Ha Manh Tran (in alphabetical order).

Doc ID: D9.5-v1.0.doc

AMENDMENT HISTORY

Version	Date	Author	Description/Comments
V0.1	September 22, 2009	Marinos Charalambides	First version - ToC
V0.2	December 2, 2009	See authors list	Initial draft
V0.3	December 4, 2009	Fabio Hecht	Added P2PCOST2 report
V0.4	December 20, 2009	See authors list	Second round of inputs
V0.5	December 28, 2009	Marinos Charalambides	Proof reading and editing
V1.0	December 31, 2009	Marinos Charalambides	Final editing, final version

Table of Contents

(This page is left blank intentionally.)

Executive Summary

Autonomic management has been receiving significant research interest from both academia and industry in the recent years and has emerged as an appealing solution to the complex task of managing modern networking environments. Under this paradigm, the management system can take decisions in accordance to predefined goals and emerging conditions in the absence of human administrators. The set of properties that such a system should exhibit are known as the self-* properties. These are self-awareness, self-configuration, self-optimisation, self-healing and self-protection.

Work Package 9 (Autonomic Management) of the EMANICS project aims at researching the self-* properties of autonomic systems and proposing solutions to the many challenges surrounding autonomic management. Through research integration and collaboration among WP9 partners, the following three tasks have been undertaken during the third phase of the project:

- *Task 1: Autonomic management of fixed networks and ubiquitous environments*. Key issues addressed by this task involve the automation of service recovery and resilience; autonomic management elements for intelligent service composition and end-to-end service delivery; federated management between administrative domains; self-healing and protection mechanisms in resource constrained ubiquitous environments.
- *Task 2: Policy-based autonomic management*. The focus of this task is on the use of the policy-based paradigm for automating the decision making process in the context of fault recovery; the self-configuration of devices with limited capabilities; and the specification of a policy continuum.
- *Task 3: Peer-to-peer approaches for autonomic network management*. Key issues addressed by this task involve misbehaviour and optimization in collaborative environments; alignment of incentives; distributed P2P protocols and swarming algorithms.

This deliverable (D9.5), builds on collaborative work performed during the previous phase of the project. The six activities presented in this report describe the work carried out during the last phase and conclude the research work in WP9. Each activity addresses various aspects of one or more of the above tasks as they have several overlapping areas being targeted by the WP9 partners' joint research efforts.

1 Introduction

Managing large scale systems composed of vast numbers of heterogeneous devices that communicate with each other by means of a wide variety of physical media and communication protocols is a very complex and expensive task. The problem is exacerbated due to the dynamic and volatile nature of many of today's networks where users, in addition to roaming freely, can typically enter and leave the network without any warning, thus making many of the network services intermittent. The IT management experts never seem to be enough to maintain control over fixed networks let alone dynamically changing networks. This is the reason why autonomic management is such an appealing solution to the ever growing complexity of management systems. With autonomic management the burden of managing a system is passed from the administrators to the system itself. This means that the system becomes self-managed, or self-governed, leaving to the administrators the responsibility of steering the system's overall behaviour through desirable high-level objectives. However, achieving autonomic management requires research work to encompass different areas due to the extensiveness of the autonomic management field. In this light, WP9 has set up six activities to address the many challenges of autonomic management.

Networked services are nowadays built on large scale, heterogeneous and volatile infrastructures. Despite the advance of technologies, faults still frequently happen in a large-scale, human administrated IT environment. One of the key challenges that administrators and operators are facing today is how to quickly find recovery solutions to effectively eliminate faults and reinstate the impacted services. To address this challenge, Section 2 builds on previous joint efforts and investigates automated fault resolution and recovery mechanisms. The former involves using peer-to-peer technologies and artificial intelligence techniques such as case-based reasoning for fault resolution. The latter applies AI-based automated planning methods to assisted administrators by fault recovery process planning. Furthermore, techniques for optimizing the performance of services in IP-based networks have been developed. In the case of faults, such as link failures for example, the network can be reconfigured in a flexible manner.

The immediate future is depicting a situation where the Internet will be the common place where different autonomous entities will be sharing the same set of resources. Virtual networks, corporate computing domains, and services spanning different network provider domains are examples of such a situation. To effectively achieve the business objectives, a kind of federation needs to be established between autonomous management entities. Section 3 extends previous efforts regarding an architecture enabling the establishment of a federated management environment through governance and negotiation functionalities. Methodologies for the orchestration of autonomous entities as well automated resource negotiation models are described in this section.

In recent years, policy-based management has been proposed as an enabling technology for autonomic management. Polices offer a clean approach to differentiate the functionality of a system from the rules employed to lead the system to a desired state. A policy-driven system can monitor itself and its surroundings so as to adapt its functionality according to the current conditions and the system objectives. This provides autonomic behaviour in the form of a closed feedback loop that can be applied locally to individual system components or hierarchically to sets of cooperating components. Section 4 investigates the challenges when using the policy paradigm to realize autonomic management systems (AMSs). The existing core components of the policy-based paradigm are revised to meet the requirements of autonomic management so that policies can support control loops, and thus self-* capabilities, implemented by AMSs. Furthermore, a policy continuum separates policies into multiple levels of concern that jointly drive an autonomic management system architecture.

Another dimension of WP9 concerns the management of wireless networks consisting of small devices. These can form pervasive systems that are expected to perform in a vast number of environments, ranging from urban to rural, with different requirements and resources. They are often mobile and the environment and application requirements may change dynamically requiring flexible adaptation. Section 5 describes self-healing techniques for handling faults in wireless sensor networks with which faults can be masked or isolated. Security issues are also considered so that misbehaving nodes or attacks on the network can avoided. Furthermore, optimization policies are used so that the lifetime of handheld devices can be improved.

The constantly increasing data rates available in the core networks of ISPs and the increasing number of concurrent flows running through the network became a challenge for traditional centralized IP traffic analysis systems. The design of distributed and autonomic mechanisms suitable for real-time IP traffic analysis on (very) high-speed network links is the subject of Section 6. The shortcomings of the hierarchical approach for a distributed flow records storage platform identified in the previous phase are addressed with a "flatter" peer-to-peer (P2P) organization approach. In addition, traditional centralized relational databases and how well they can cope with high amounts of traffic data are investigated in this section.

The popularity of devices with IP connectivity, both stationary and mobile, combined with growing reachability of such networks, increases the demand for high-quality video streaming over the Internet. In contrast to centralized approaches the P2P technology can reduce bandwidth and management cost for the distributor and add scalability as the number of users watching the stream increases. Current proposed incentive mechanisms are either susceptible to many kinds of Sybil attacks, fail to distinguish between the incentives of peers, or do not consider lag minimization explicitly. Section 7 describes a protocol for Sybil-aware directed swarming for the distribution of live content. The approach is resistant to certain classes of known mechanisms for "lying peers" and Sybil attacks.

Purpose of the Document

The purpose of this document – deliverable D9.5 – is to provide a detailed description of technologies and solutions surrounding the challenges of autonomic management. The material presented in deliverable D9.5 is the outcome of research integration and collaboration among partners of the EMANICS Network of Excellence Work Package 9 (WP9: Autonomic Management). As intended by the EMANICS partners, the work presented in this deliverable enhances and builds on research work developed during the second phase of the project which was presented in deliverable D9.4. Additionally, this document provides evidence that WP9 has succeeded in building strong collaborative research links between partners, which is imperative to achieve the goals set up by the EMANICS project.

This document presents the research results produced by the six activities approved during the open call of EMANICS Phase 3. Four activities are the continuation of research carried out during Phase 2, while two new activities have been added.

2 Automated Service Management using Emerging Technologies (ASMET3)

2.1 Introduction

One of the key challenges in today's IT environments is automatically determining and applying recovery solutions in response to emerging faults. ASMET3 extends previous efforts in this area by considering peer-to-peer technologies and artificial intelligence techniques. The knowledge discovery and fault resolution activities focus on a distributed case-based reasoning system that exploits various on-line knowledge sources and reasoning capabilities in a decentralized, self-organizing platform provided by peer-to-peer technologies. It also looks into reasoning methods that can deal with semi-structure fault data in order to provide useful information for fault resolution. The focus is on a reasoning method that can take advantage of the textual description of fault data such as bug reports extracted from bug tracking systems.

For the automated planning of fault recovery process, ASMET3 applied the Hierarchical Task Network (HTN) based planning paradigm to compose the plan for the fault recovery activities. The advantage of the HTN planning is that, its operational model resembles the problem-solving procedures of human operators with stepwise refinements on the granularity of the solution; each refinement step allows the integration of formal and empirical management knowledge. The capabilities of knowledge integration and constraint processing empowers HTN planner with capabilities to solve real-world planning problems. Comparing to other planning paradigms such as state-space or plan-space planner, it has the advantages of efficiency and flexibility in finding of plans and it has been the most applied planning paradigm in solving the real-world planning problems.

This activity also investigated ways for optimizing the performance of services in IP-based networks. More specifically, ASMET3 considered techniques to control traffic routing and subsequently support quality of service. In the case of faults, such as link failures, methods for calculating and configuring alternative routes have been developed so that the impact on affected services can be minimized. Network (re-) configuration is achieved with policies in a flexible manner.

This report is organised as follows: Section 2.2 gives a review on the ASMET architecture; Section 2.3 provides a detailed discussion on the knowledge discovery components; in Section 2.4 we provide insight of the automated planning components, detailed views are given on the mechanisms behind those approaches; in Section 2.5, a novel approach to the policy-based network reconfiguration for fault remedies is presented.

2.2 An Overview of ASMET Architecture

In this section we intend to give an overview on the ASMET architecture as a recap of the previous contributions. The ultimate goal of the ASMET project is to provide administrators of networked services a tool to facilitate the decision-making process during the fault recovery activities. The advanced reasoning and searching capabilities on both knowledge discovery level and workflow level can assist operators by exploring and integrating the distributed recovery knowledge into fault recovery operations in an automated manner and thus to increase the efficiency of the recovery operations by reducing the Mean-Time-To-Repair (MTTR) value.

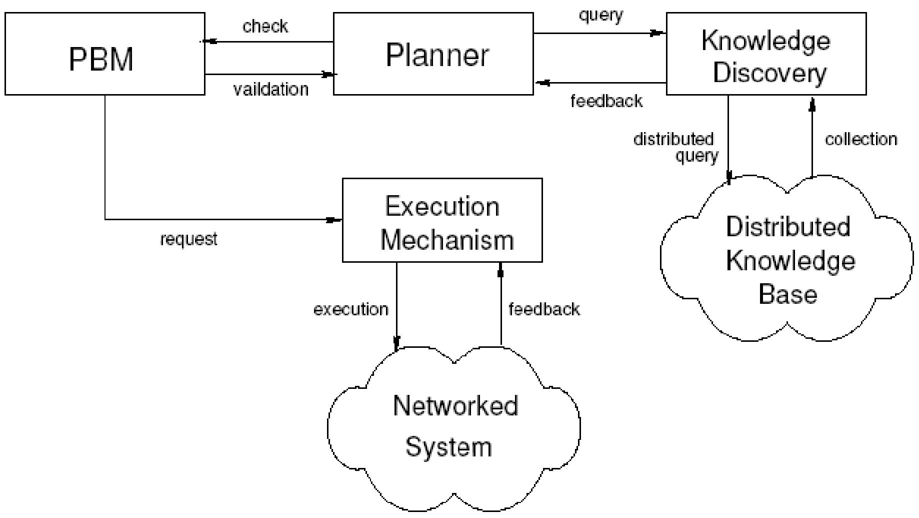


Figure 2-1. An overview on the ASMET architecture

Figure 2-1 shows a component-based overview on the ASMET architecture; it comprises of three key components: planner, knowledge discovery component and a policy based management system.

The planner is designed to searching for fault recovery plans according to the current state and the designated goal state of the observed service. The AI-based planning algorithm utilises those information as input for the planning process. With the recovery knowledge provided by the knowledge discovery component, the algorithm calculates the detailed, executable workflows for recovery operations. The automatically composed plan is then validated by the policy based management system in order to see if the plan conforms to relevant management policies. Upon a positive validation result, the execution plan will be delegated to the execution mechanism for the implementation on the real system.

The purpose of the ASMET architecture is to apply the advanced AI technologies, such as automated planning, distributed case-based reasoning and knowledge-based management approach, to alleviate the complexity of decision-making process regarding the fault recovery of networked services, which is usually posed on the system operators.

2.3 The Principle for the KD Component – Probabilistic Reasoning

The knowledge discovery component (KB) is designated to operate on a set of distributed recovery knowledge bases. Based on the case-based reasoning (CBR) technique, the KB is supposed to analyse user’s requests and crawl various knowledge sources for fault recovery solutions. In this section, we provide a deep insight on the core mechanism of the CBR engine in the KB component – the probabilistic reasoning.

Two prevailing reasoning approaches in CBR systems are transformational reuse [1][2] and derivational reuse [3][4]. The salient characteristic of these approaches is to avoid choosing an identical solution from the retrieved cases, they instead resolve a problem by mapping the source and target case structures using transformational rules, or by replaying a problem-solving process using inference traces captured before. These approaches, however, demand substantial knowledge sources, complicated case representation and processing to which many problem domains cannot afford [5][6]. Applying these approaches to the communication system fault domain, shortly the fault domain, is difficult because fault cases contain semi-structured data and present a variety of problems and solutions without a guarantee for the existence of correct solutions. The transformational approach requires experienced problems structurally similar to the problem or the derivational approach requires adapting traces captured before. Other proposed approaches identify the problem into problem categories [7][8][9][10], or seek promising solutions among experienced solutions [11][12]. Our method aims to provide promising solutions quickly with less support from operators.

The motivation for applying a probabilistic reasoning (PR) method to CBR systems comes from the demand of making decision under uncertain situations. This method built on probability theory has been used either for indexing cases in case retrieval or for inferring cases in case reuse. We study the application of PR methods to the reasoning engine of CBR systems for dealing with the limitation of knowledge sources. In particular, an engine rests on an incomplete case database to decide which cases potentially resolve the problem, and which case properties properly infer facts related to the possible solution. We have seen several reasoning methods that broadly fall into two categories:

- *Using the entire case database* [7]-[10]: This method constructs a probabilistic model based on the whole case database; e.g., determining the problem distribution of a case database based on case properties. The model can then be used to conjecture the class of the problem. The method usually requires a large case database to build the model as accurate as possible. It is appropriate for classifying problems in problem domains that possess a small set of problem categories. In several studies, this method takes advantage of automatic learning algorithms to establish the probabilistic model that can also be used for indexing and retrieving cases in CBR systems.
- *Using the partial case database* [11][12]: This method is suitable for selecting solutions (diagnosing and searching) in problem domains with a variety of problems and solutions. The method relies on a small number of relevant cases obtained by case retrieval and provides more concrete solutions for problems. A probabilistic model can be built up by using the typical properties of the retrieved cases and then be used to select or infer promising solutions; e.g., determining case properties such as symptoms and causes influencing a problem significantly. The probabilistic model can be established by the experts and automatic learning algorithms.

This part briefly revisits Bayesian formulas that will be used in this paper. Considering H as a hypothesis and $e_n = e^1, \dots, e^n$ as a sequence of evidence pieces obtained from a scenario with an assumption that $e^1, \dots, e^n$ are independent from each other. The posterior probability for the multi-evidence scenario is derived by the conditional probability formula:

$$P(H \mid e_n) = \frac{P(H)P(e_n \mid H)}{P(e_n)} = \frac{P(H)\Pi_{k=1}^{n} P(e^k \mid H)}{P(e_n)} \qquad (2.1)$$

where $P(e_n \mid H) = \Pi_{k=1}^{n} P(e^k \mid H)$, *since evidence pieces* e^k are independent from each other, $P(H)$ is the prior probability of hypothesis H and $[P(e_n)]^{-1}$ is determined by the requirement $P(H \mid e_n) + P(\overline{H} \mid e_n) = 1$. This formula indicates that the belief of hypothesis H upon receiving e^k can be computed by the previous belief P(H) and the likelihood $P(e^k \mid H)$ that e^k will materialize if H is true.

This formula also serves the purpose of prediction and diagnosis. The prior probability P(*H*) measures the *predictiv*e support accorded to H by the background knowledge, while the likelihood $P(e_n \mid H)$ represents the *diagnosti*c support given to H by the evidence pieces actually observed. The posterior probability $P(H \mid e_n)$ indicates the strength of belief in a hypothesis H based on the previous knowledge and the observed evidence pieces e_n.

2.3.1 Reasoning Method

We propose a probabilistic reasoning method that uses the partial case database for case reasoning in DCBR. Since DCBR has already included a case retrieval method that obtains relevant cases from peers' entire case database [14], we only focus on a case reasoning method that infers promising solutions from a set of the retrieved cases in this study. The solution selection method tends to be more successful for a small, well-constrained set of problems [15]. Moreover, this method can obtain better processing time by working with a small set of cases.

Our PR method contains two *rankin*g and *selectio*n processes. The first process operates on symptoms to figure out cases showing the same symptoms as the problem's. This process aims to narrow down the scope of the problem by providing a smaller set of promising cases. Note that complicated cases comprise many symptoms and diagnoses, reducing a number of cases leads to a smaller number of symptoms and thus lower computation cost. Formally, given the problem C_p and the relevant cases C_r acquired by case retrieval, both C_p and C_r share a set of common symptoms. The process estimates their similarity by using the k-Nearest-Neighbor algorithm [16] with a similarity function defined as follows:

$$sim(C_r, C_p) = \sum_{i-1}^{t} w_{ri} w_{pi} \tag{2.2}$$

where t is number of common symptoms, and w_{ri} and w_{pi} are the weight values of these symptoms in C_r and C_p respectively. The similarity function considers two factors: the number of common symptoms between two cases and the significance of these symptoms in each case.

The second process aims at predicting promising solutions for the problem using Bayesian computation. Given a set of cases *C*, where a case *C*r contains a set of symptoms $\{S_1,\dots,S_k\}$ and a solution, we assume that solutions in C as a set of exhaustive and mutually exclusive hypotheses $\{H_1,\dots,H_n\}$, and that any symptom is the result of a diagnosing probe, e.g., a ping probe provides either the high probability of success or the low probability of success (i.e., failure). The problem contains a set of symptoms $\{S_1,\dots,S_h\}$ without a solution (note that cases and the problem can share the same symptoms). Thus, the puzzle is to find the highest conditional probability of the hypotheses $P(H_i | S_1,...,S_n)$ with i =1,...,*n*.

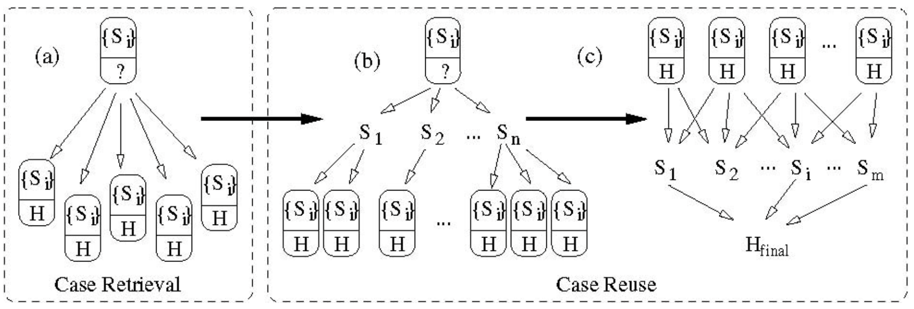


Figure 2-2. {Si}, H, ? and H_{final} denote a set of symptoms in a case, a hypothesis, the unknown hypothesis and the promising hypothesis respectively. (a) Case retrieval based on the evaluation between the problem and cases through vectors, (b) Case ranking based on the evaluation between the problem and cases through symptoms, (c) Case selection based on the correlation among cases and the problem through symptoms.

Considering a set of exhaustive and mutually exclusive hypotheses $\{H_1,...,H_n\}$ and $\{S_1,...,S_h\}$ as a set of evidence pieces (or symptoms) obtained from the problem with an assumption that $\{S_1,...,S_h\}$ are independent from each other.

Applying Eq. 2.1, we obtain:

$$P(H | S_1,...,S_h) = \frac{P(H)P(e_n | H)}{P(e_n)} = \alpha P(H_i) \prod_{i=1}^{h} j_i \qquad (2.3)$$

where $P(S_1,...,S_h | H_i) = \prod_{j=1}^{h} P(S_j | H_i)$, since S_j are independent from each other, $P(H_i)$ are the prior probabilities of hypothese, and $\alpha = [P(S_1,...,S_h)]^{-1}$ is determined via the requirement $\sum_{i1}^{n} P(H_i | S_1,...,S_h) = 1$.

In case the evidence set S contains a new evidence S_{new} (e.g. the problem has a new symptom), updating the new evidence piece first computes $P(H_i | S)$ and the uses $P(H_i | S_1, S_{new})$ as follows:

$$P(H_i | S, S_{new}) = \frac{P(H_i | S)P(S_{new} | S, H_i)}{P(S_{new} | S)} = \beta P(H_i | S)_{new_i} \qquad (2.4)$$

where β determined by the same method as α, $P(H_i | S)$ is computed as previously, and $P(S_{new} | H_i)$ is determined by experts. The algorithms simply reflect the above discussions. Algorithm 1 iterates over cases (2), finds common symptoms (3) and accumulates weight values (6, 7) before ranking cases in the resulting set (8). Algorithm 2 iterates over hypotheses (2), initializes the prior probability with a value $\frac{1}{n}$, where n is the number of hypotheses (3), computes and normalizes the posterior probabilities (4, 5, 6, 7) before ranking cases in the resulting set (8).

```
Algorithm 1: Case Ranking
  Input: $C$: a set of cases $C_r$
  $C_r$, $w_r$: sets of symptoms & weight values
  $C_p$, $w_p$: sets of symptoms & weight values
  Output: $R$: a ranked set of cases

1 $R \leftarrow \emptyset$
2 for each $C_r \in C$ do
3   $S = C_r \cap C_p$
4   if $S \neq \emptyset$ then
5     $T_r = 0$
6     for each $S_i \in S$ do
7       $T_r = T_r + w_{ri} w_{pi}$
8     insert $C_r$ in $R$ in the order of $T_r$
```

```
Algorithm 2: Case Selection
  Input: $C$: a set of cases $C_r$ & solutions $H_r$
  $V$: a list of probability values $V_r$
  $C_r$, $w_r$: sets of symptoms & weight values
  $C_p$: sets of symptoms $\{S_1, \ldots, S_h\}$
  Output: $F$: a final set of solution cases

1 $F \leftarrow \emptyset$
2 for each $H_r \in C$ do
3   $V_r = \frac{1}{n}$
4   for each $S_i \in C_p$ do
5     $V_r = V_r w_{ri}$
6 for each $V_r \in V$ & $C_r \in C$ do
7   $V_r = V_r \|V\|^{-1}$
8   insert $C_r$ to $F$ in the order of $V_r$
```

Figure 2-3. Case ranking and selection algorithms.

Example: We have a set of cases with the following solutions and symptoms related to the *connectio*n *failur*e problem:

- *H*1 = Checking firewall software for blocking connections
    - *S*1 = Desktop keeps disconnecting from the Internet
    - *S2 = Desktop and Laptop keeps connecting from the router*
    - *S3 = Connection usually goes really slow*
    - *S*4 = Connection is fine before updating the firewall software
    - *S5 = Router is WHR-HP-G54 and wireless adapter is Linksys WMP54G*

←

- *H*2 = Reinstalling networking components (TCP/IP)
    - *S1 = Desktop completely stops connecting to the Internet*
    - *S2 = Laptop can connect to desktop and the Internet*
    - *S3 = Desktop disconnects to laptop and D-Link router with a limited connectivity*
    - *S*6 = Desktop uses an Etherlink 10/100 PCI card and laptop uses a wireless adapter
    - *S*7 = Registry was damaged on desktop few days ago

i.

- *H3 = Checking router configuration for the IP address range*
    - *S*1 = Desktop cannot connect to a router and the Internet
    - *S*2 = Laptop connects to the router and the Internet
    - S4 = The firewall software is often updated on those machines
    - S8 = Desktop gets error message of address already used when renewing

The following table presents the weight values of symptoms to the solutions (note that updated weight values are not bold). This table is for demonstration, we only need weight values related to the problem's symptoms for implementation:

	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
H_1	**0.1**	**0.1**	**0.25**	**0.448**	**0.001**	0	0.1	0.001
H_2	**0.1**	**0.1**	**0.3**	0.001	0	**0.048**	**0.45**	0.001
H_3	**0.25**	**0.245**	0	**0.005**	0	0.05	0	**0.45**

In order to fulfil the requirement of exhaustive and mutually exclusive hypotheses, we examine a set of hypotheses and H_1, H_2 *and* H_3, and ignore other hypotheses; i.e., the conditional probability of other hypotheses is 0. This, however, can be a problem in practice if the examined set of hypotheses does not contain the desired hypothesis. We also consider a set of evidence pieces $S_1, \ldots, S_8$ obtained by distinct probes independent because the effect of a probe to other evidence pieces is minor, and evidence pieces can only be correlated if probes are not distinct; e.g., if a connection failure occurs, symptoms collected by the *pin*g and *ft*p probes can be correlated. Hypotheses possess the same prior probabilities P (H_i) = (0.33, 0.33, 0.34), and the problem contains the following symptoms:

- H =?
- ← *S1 = Desktop gets connection failure*
- ← *S2* = Other machines still connect to routers and to the Internet,
- ← *S4* = Desktop updated the firewall software two days ago.

By applying Eq. 2.3, we obtain $P(H_1 | S_1, S_2, S_4)$ = (0.8719, 0.0019, 0.1261) with i =1, 2, 3. The result indicates that the chance of firewall software blocking connections is 87.19% given the symptoms of the problem. Intuitively, a solution is deduced by an incomplete set of symptoms; solutions likely share a subset of symptoms. Bayesian computation distinguishes those solutions by the significance of symptoms in a case and the significance of symptoms among cases.

2.3.2 Evaluation of Probabilistic Reasoning

We have created a dataset to evaluate the performance of our reasoning method. The dataset contains bug reports crawled from four bug tracking systems (Trac, Bugzilla, Mantis, and Debian) and stored in a unified data model [17]. For the ranking process, we have generated a set of 50 queried problems. Each queried problem possesses a set of significant keywords and symptoms extracted from the dataset. After retrieving similar bugs from the dataset, see Figure 2-2a, we perform the ranking process on the retrieved bugs, and then verify the precision of the process. The precision rate is measured by the ratio of the number of correct bugs obtained to the total number of bugs obtained. A correct bug shares a high number of common symptoms with the queried problem. We ignore the recall rate because all of the retrieved bugs from case retrieval were similar to the queried problem already. We use a threshold θ to obtain bugs, where θ is determined by the average value of ranked values of bugs (*Tr*) in the ranked set, see Algorithm 1. The small dataset for experiments contains 71,450 bugs.

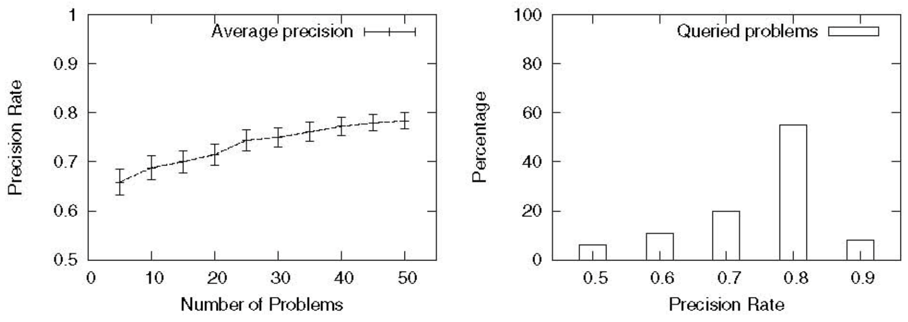


Figure 2-4. Average precision by number of problems and problem distribution by precision

Figure 2-4 depicts the performance of the ranking process based on the precision metric. The line graph increases from 0.65 to 0.78 over 50 problems because correct bugs dominate in all bugs obtained for each problem. The criteria of correct bugs therefore have an impact on this result. The ranking process achieves a precision rate higher than 0.7 on average. Since the resulting bugs from the ranking process contain a high number of common symptoms, the selection process can effectively use them for reasoning. The bar graph also characterizes 83% of problems receiving precision values higher than 0.7 and 17% of problems receiving precision values lower than 0.7. We learned that the bug dataset is wide and diverse in scope and only a few cases specifically focus on the same problem in the fault domain, thus it is difficult to apply the dataset for the selection process.

We have created another dataset to evaluate the selection process because the crawled bug dataset currently requires extra work on symptoms extraction and problem classification. This dataset comprises problem scenarios published at a networking forum [18] that concentrates on problems in a small-scale network such as a home network or an office network. This dataset contains 60 *connectio*n *failur*e cases resulting from various causes: (i) *hardwar*e including network card malfunctioning, router malfunctioning, bad cable, etc (13 cases), (ii) *softwar*e including firewall blocking, bad network card driver, bad networking component, router upgrading, etc (22 cases), and (iii) *configuration* including bad router setting, bad network setting, bad security setting (25 cases). Cases contain a hypothesis and a set of symptoms with weight values. We have generated three sets of queried problems with one (S_1), two (S_2) and three (S_3) common symptoms extracted from the dataset.

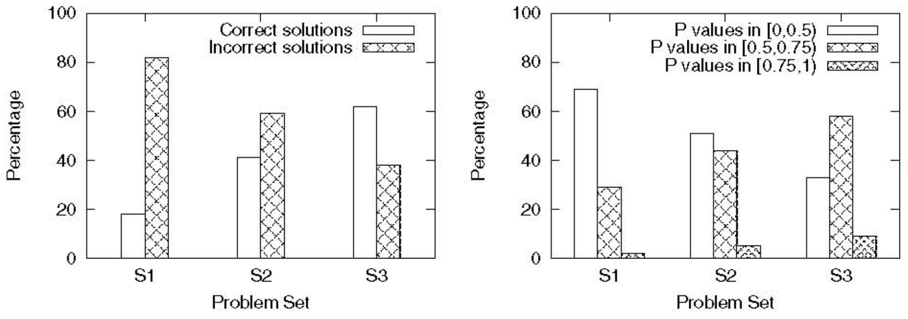


Figure 2-5. Solution distribution by number of problems and probability distribution by correct solutions for three problem sets (P denotes probability).

We perform the selection process on the queried problems, and verify the proposed solutions with *correc*t or *incorrect*. The left bar graph in Figure 2-5 indicates that, with the increasing number of common symptoms in S_1, S_2 and S_3, the distribution of incorrect and correct solutions are inverted. S_1 receives 82% of incorrect solutions and S_3 receives 38% of incorrect solutions. We observed that while incorrect solutions in S_1 are caused by ambiguous information (e.g., vague symptoms with high weight values), incorrect solutions in S_3 are caused by confusing information (e.g., two salient symptoms with high weight values). Thus assigning weight values to symptoms crucially affect the outcome of solutions.

We investigate the probability distribution of correct solutions in S_1, S_2, and S_3 to learn further the dataset and the queried problems. The right bar graph in Figure 2-5 shows that the number of solutions with $\mathrm{P} \geq 0.75$ is very small compared to the number of solutions with *P<0.75*. A solution with more common symptoms receives high probability due to the accumulation of weight values. A solution with fewer symptoms still receives high probability if these symptoms are distinct. A solution receiving low probability indicates that either the case or the problem provides too general information such as vague symptoms. Reducing these vague symptoms can make the dataset better, thus ameliorating the performance of the selection process.

2.4 Recovery Planning based on the Hierarchical Task Network

A fault recovery process usually involves a sequence of actions, which can change the state of a defected service stepwise toward a desired operational state. By manual recovery operations, a human administrator often faces series of intricate decisions to make, such as, which recovery actions are applicable to the current situation? Do the selected actions contribute to the final recovery goal? How could actions to be properly scheduled in an operational environment, where applicable resources are limited? etc. To entangle the matter further, she has to determine if the composed plan conforms to the applicable policies and constraints to the impacted services.

As the IT services today are becoming more complicated and interdependent, it is an arduous and time-consuming process for a operator to make right recovery decisions efficiently in a highly dynamic operational environment. To alleviate those problems, the research of ASMET project is designated to provide architecture to offer a machine-aided decision-support service tailored to assist the fault recovery process. Specifically, this architecture is designed to confront with two challenges that today's service operators are facing:

- How to find the knowledge that is relevant to the fault recovery process? As service recovery is a knowledge-intensive process, an efficient knowledge discovery method is undoubtedly an integral part for the entire recovery operation.
- How to effectively apply the discovered knowledge into the recovery process? We need a supporting tool to integrate and to correlate all fault-recovery relevant knowledge into the planning process.

The previous section on distributed reasoning provides insights on the automated knowledge discovery mechanism to address the first research challenge. In this section we focus on the automated recovery planning mechanism.

Based on the AI-based automated planning techniques, the recovery planning algorithm is designed to integrate and to reason upon the recovery knowledge in order to automatically compose feasible recovery plans. We particularly concentrate on the planning paradigm based on hierarchical task networks. Our choice is based on multiple reasons [19] : i) the hierarchical way of problem-solving technique applied in the HTN planning simulates how human experts solve practical planning problems; ii) it allows to encode and to integrate problem-solving recipes into planning process; iii) HTN-based technique is one of the mature planning approaches that has been researched by the AI community for many years; it is the most applied planning technique to solve mission-critical and complex real-world problems [24].

2.4.1 Hierarchical Task Networks

The basic idea behind the Hierarchical Task Network (HTN) [20] planning is the refinement of the planning tasks based on the decompositions of those tasks at different hierarchical levels. In the definition of HTN planning, there are three kinds of tasks: decomposable tasks, primitive tasks and goal tasks. As the name suggests, the decomposable tasks are those tasks that could be refined into more detailed levels and could not be directly executed. The primitive tasks are directly executable operations.

```
Algorithmic Description of HTN Planning

Input: $P$: planning problem definition
       $T$: Task Network, where $t_{primitive}, t_{decomposable} \in T$
       $M$: set of methods
Output: $\pi$: sequence of primitive tasks
$\pi \leftarrow \emptyset$
Choose $t \in T$, where $t_{post-condition}$ satisfies $s_{goal}$, $s_{goal} \in P$
$\pi = \pi.t$
while True do
   if $t \in \pi \wedge \forall t \in T_{primitive}$ then
      resolve conflicts in $\pi$
        if Success then
          return $\pi$
        else
          return Failure
   Choose $m \in M$
   Decompose $t$ with $m$
```

Figure 2-6. General HTN-Planning Algorithm

The goal tasks describe the desired final states of the target system. Comparing to other planning paradigms, the HTN planner introduces a new element called *method* into the planning procedures. A *method* is a construct which contains the refinement of more specific operations organised in a form of a partially ordered network. Operations could include atomic actions or other methods. The planning operation is an iterative process of decomposing high-level tasks into lower level operations until there is no more decomposable tasks. The planning process maintains a tree-like structure, where nodes are decomposable tasks and leaves contain atomic operations. The main purpose of HTN-based planning is not to achieve a defined goal, but rather to find out how to perform a given set of atomic or decomposable tasks. Figure 2-6 shows a algorithmic description of HTN planning paradigm.

The HTN-based planner requires as input a definition of problem description P. The planning problem *P* could be mathematically modelled as a tuple: $P = (\sum, s_0, g)$, where $\sum$ is a state transition system and $s_0 \subset S$ denotes the current state, it is a subset of the state description *S*, and $g \subset S$ is the desired goal state. The solution to the planning problem *P* is a set of actions *A*, which transits the system $\sum$ into the desired state g. The transition system $\sum$ possesses a transition function γ, where $\gamma : A \times S \rightarrow S$, meaning the transition function γ changes the current system state into a new state based on the operation of an action from *A.* A method M to decomposition of tasks is defined as follows:

$$M = (M_{ID}, M_{task}, M_{pre}, M_{network}) \tag{2.5}$$

where M_{ID} denotes a unique name of the methods which distinguishes this methods from others., M_{task} describes this decomposable task, M_{pre} is a set of literals, which describes the pre-conditions of this tasks and $M_{network}$ contains a partial network that includes the subtasks of this methods. Note that the sub-tasks could contain another set of decomposable tasks or primitive tasks or both. A method is totally ordered if its corresponding subtasks are totally ordered. Figure 2-6 gives an algorithmic description of the HTN based planning. Note that during the task decomposition process, attentions should be given to avoid conflicts between refined tasks and other existing tasks, the 5th line in the table checks the consistency of the plan after each refinement steps. With minor modification of the planning methods, constraints could also be applied to the HTN planning, those constrains include ordering constrains, resource usage constrains, time constrains and so forth. The capability to handling different types of constraints greatly extended the applicability of the automated planner to solve real-world planning problems.

2.4.2 Information Models for Recovery Planning

The operation of the automated recovery planning process is based on the diverse input knowledge. In this section, we show the corresponding information models of such planning knowledge.

2.4.2.1 Recovery Method

Figure 2-7 shows an information model for the recovery method, which could be applied by the HTN-based planning mechanism for the purpose of the recovery task refinement. The ***Method*** class is associated with preconditions and effect. The ***Precondition*** class defines the applicability of a particular recovery method to those situations, under which the recovery method can be effective. The ***Effect*** class denotes the post-conditions after the execution of the tasks included in this method. It records changes of the system states which are caused by the implementation of a particular action. Both Precondition and Effect classes could be described using set of logical expressions.

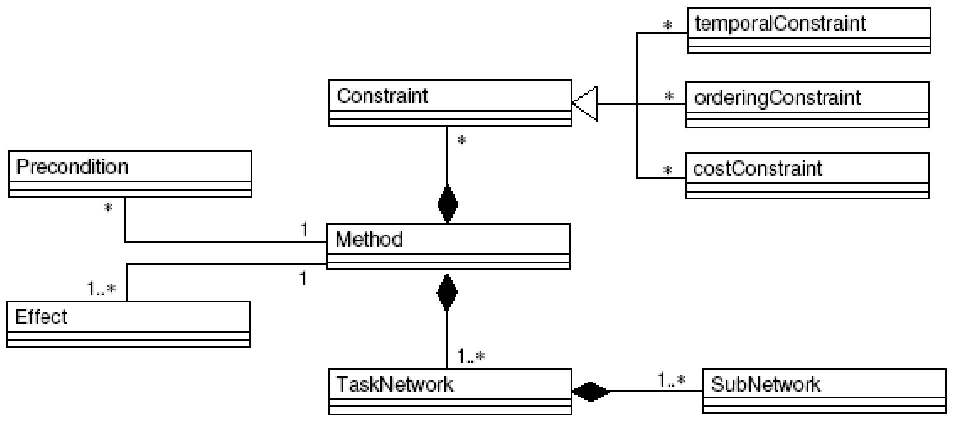


Figure 2-7. Information Model of Recovery Method

The ***TaskNetwork*** class describes a set of tasks that could be applied in a particular method. A task is called primitive task if it is non-decomposable, otherwise it is called a compound task. A TaskNetwork can be represented as an acyclic digraph $w=(U,E)$, where ***U*** is the node set and ***E*** denotes the set of edges in the graph. Each node $\{u \mid u \in U\}$ contains a task t_u. The network ***w*** is a primitive task network if all $t_u, u \in U$ are primitive, otherwise it is a non-primitive task network. The edge of the graph is determined by the ordering constraint that we will discuss shortly.

The ***SubNetwork*** class represents the sub-tasks in a task network. A complex non-primitive task network may be comprised of multiple sub-tasks, which could be further decomposed into next level. Therefore, task networks can be defined recursively to include multiple levels of decompositions. The advantage of using ***TaskNetwork*** is that, it provides a way to encode the recovery knowledge in the structured data, which allows the integration of management knowledge into planning process. The following example gives an instance as an example of the task network.

Example: Failover method is frequently applied to offer business continuity during the fault recovery; it involves series of actions in order to complete the failover operation. In this example, we illustrate this operation in terms of HTN planning method.

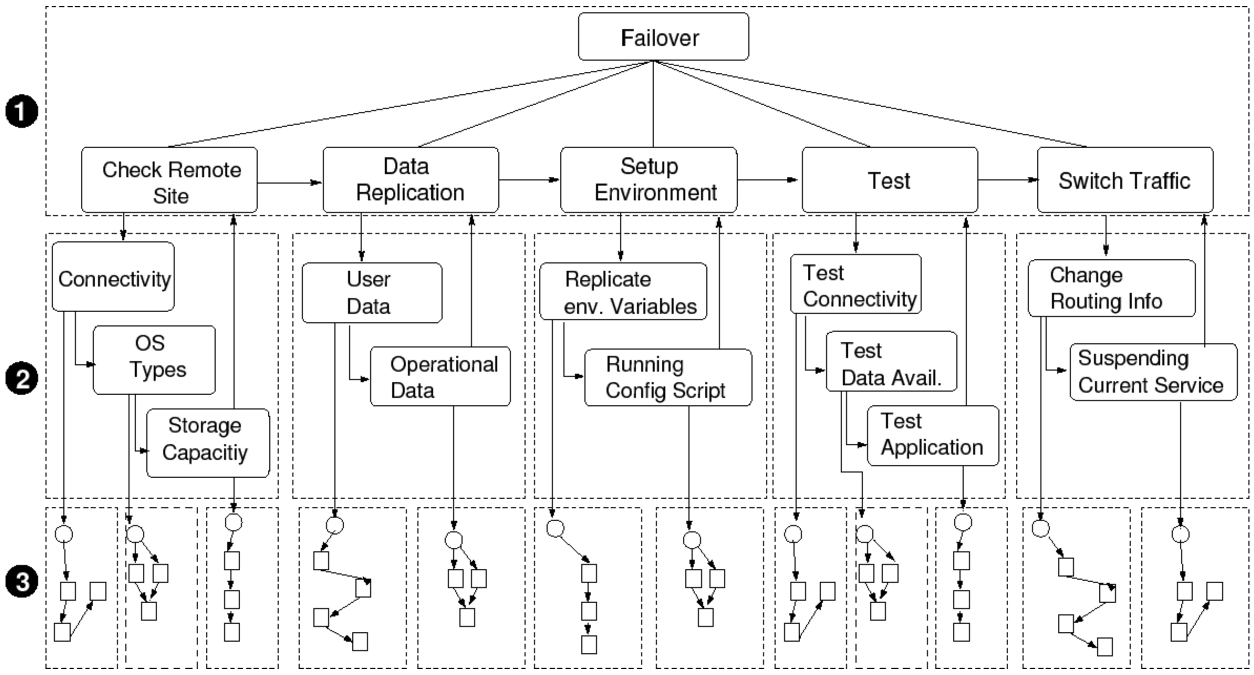


Figure 2-8. A Task Network Example

Figure 2-8 shows a simplified process of failover operation in form of the HTN planning task network. The method ***Failover*** consists of several sub-task networks: checking the characteristics of the remote failover site; replicating data to the remote site; setting up operational environment for the service: testing and validating the service and finally, switching the traffic to the failover site. All of the sub-tasks, which are included in the failover method, are non-primitive ones, because those tasks can be further decomposed into more detailed levels. The boxes with dashed lines denote different levels of task networks.

The second level shows the further decompositions of the sub-tasks. For example, checking remote failover site task is comprised of more detailed actions including: checking of the connectivity of the remote failover site to ensure the accessibility; determining the type of operating system running on top of remote site and ensure that the storage capacity meets the requirement of the current service. Each of those sub-task networks can be further refined with detailed operations in the level 3 of the graph. To simplify the illustration, those networks are shown symbolically. The refinement operation continues until all the included tasks become primitive, which means they are directly executable on the system in terms of, for example, scripts or command-calls.

Note that Figure 2-9 only shows one possible embodiment of failover operation and failover is a commonly employed strategy to mask the service unavailability during the recovery operation, the actual recovery plan could be more complicated, since many tasks are usually required to meet the desired recovery objective. The ***Failover*** method records the recipe of how failover operation could be done.

A primitive task, which is also frequently named as the operator in planning terminologies, includes one or more executable actions. Figure 2-9 illustrates a model of such operator. The operator class is associated with set of pre-conditions and post-conditions. The pre-conditions are a set of the literals, which describe the applicability of an operator.

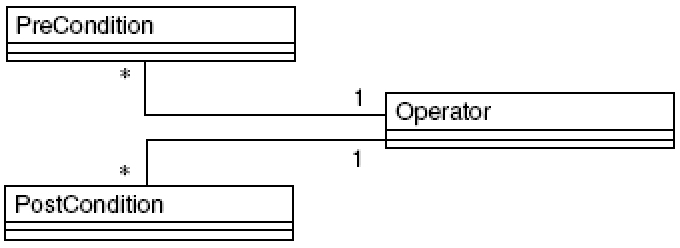


Figure 2-9. Operator Model

For example, the pre-condition for a user-data replication operation could be formulated as follows:

$$FailoverSite(y) \wedge Running(y) \wedge CapacityFailover(c) \wedge DataLocAt(x)$$

The conjunctive logic formula expresses a simplified version of pre-conditions of the replication operation. In order to do the user-data replication, it is required that there is a failover site *y* available and it is in the running state. To host the data, we have to ensure that it has enough storage capacity *c* for the incoming user-data. The current data is located at server *x*, which is the originator of the replication operation. The variables *x, y* and *c* in this example are terms that are related to certain objects, *x, y* are variables about severs and c is a variable about numerical values on the capacity. The operator to the data replication is only applicable, when all pre-conditions are satisfied.

The post conditions are similar set of literals, which describe the effects once the operation is implemented. Additional data structures keep the record on changes of those conditions. Application of the selected actions may transit the current state into a new state, for the above example, the replication operation will migrate data to location *y,* which means it is on the failover server instead of original machine. Correspondingly, this new state is included in a data structure called *add-list*, which is a list of new states, in this case, *DataLocAt(y)* is added to the this list. The changed literals are kept in a delete-list.

The resulted new state is required for further operations, for example, to setup the operational environment on the failover server, one of the essential pre-conditions for this operation is the have user data already located on that server, i.e. *DataLocAt(y).*

Predicates and variables, which describe the states of the service, are defined in the language ***L***. It is a first-order language that consists of a finite set of predicate symbols and variable symbols. Descriptions of states, including pre-conditions and post-conditions are a set of ground atoms of the first-order language ***L.***

Finally, the ***Constraint*** class is a generalization of quantifiable terms and conditions regarding time, cost as well as ordering of the specific recovery sub-tasks. The *temporal constraint* expresses the time-related conditions a recovery method must abide-by, e.g. a fallback server *A* must be used if service fault happens between 9:00 – 11:00 in order to be able to handle the huge traffic volume during the peak time, otherwise a less powerful server *B* could be applied. The *ordering constraint* determines the order of *subNetwork,* which is part of *TaskNetwork* included in the method, if u and u' are two sub-task networks, the constraints $u \prec u'$ expresses the condition that u must precede u'. Finally, the cost constraint shows the monetary conditions of this recovery method.

2.4.2.2 The Planning Knowledge Base Model

The planning knowledge base is a central repository for the information required by the planning operation. It contains objects of the fault recovery methods, actions/operators and applicable constraints. All information items, which are related to a particular service, are organized and stored in form of domain files. The operational relationship between planner and knowledge base is linked by the query operation performed by the planning algorithm to retrieve the supporting information. Additionally, the planning knowledge base also has a set of definitions on the predicates and variables. When connected as literals, they are applied to represent the states and corresponding state transformations of the service under consideration.

As a potential extension, the knowledge base model can provide interfaces to additional sources of management information, for example, from service management systems. It works as a gateway between the planning component and other management subsystems so that the planner can have an unified access for the information retrieval. For example, the distributed knowledge discovery component discussed in the previous section can be applied to enrich the content of the recovery knowledge, which is distilled from distributed bug track systems. To ensure the quality of the recovery solution plans, it is crucial that the data items in the knowledge base constantly be updated, maintained and supervised by the human administrator. To this end, the knowledge base can provide an access point for administrator to perform such management tasks. Figure 2-10 illustrates the operational relationship between recovery knowledge base and other components.

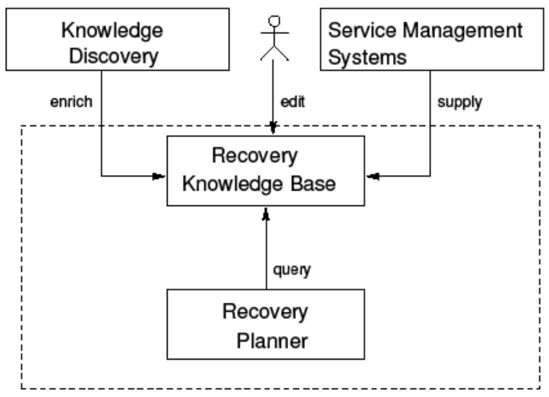


Figure 2-10. Recovery Planning Knowledge Base

2.4.3 A HTN-based Recovery Planning Algorithm

In this section, we propose a HTN-based planning algorithm to plan for the fault recovery operations. Based on the principle of the HTN planning introduced in the previous section, we start with a definition on the planning problem, which provides an operational model for the design of the algorithm.

```
INPUT: KnowledgeBase(kb), Goal Task Network(tn)
OUTPUT: Solution plan $\pi$, where $\pi = \{a_1, a_2, a_n, ...\}$

 1: function HTN(kb, tn)
 2:     if tn = $\emptyset$ then
 3:         $\pi \leftarrow$ empty plan
 4:         return true
 5:     end if
 6:
 7:     t $\leftarrow$ tn.getFirstTask()              ▷ pick one of the tasks without predecessors
 8:     active $\leftarrow$ t.getApplicableRefiners(kb) ▷ refinements with satisfied precondition
 9:
10:
11:     if active = $\emptyset$ then
12:         return false                          ▷ fail, if no refinement is applicable
13:     end if
14:
15:     performer $\leftarrow$ active.choose()                ▷ pick one of the refinements
16:     bindings $\leftarrow$ performer.computeBindings()           ▷ find its bindings
17:     b $\leftarrow$ bindings.choose()                     ▷ pick one of the bindings
18:
19:     if o $\leftarrow$ performer is an operator then
20:         kb.assert(o.getEffects(b))        ▷ simulate effects of the chosen operator
21:         tn.removeTask(t)           ▷ remove accomplished task from task network
22:         $\pi$.add(o)                            ▷ add operator to solution plan
23:         stp.decompose(t, o)          ▷ impose temporal constraints on operator
24:
25:         if stp.PC() = true then                 ▷ ensure temporal consistency
26:             HTN(kb, tn)              ▷ recursively refine remaining task network
27:         else
28:             return false
29:         end if
30:
31:     else if m $\leftarrow$ performer is a method then
32:         m.decompose(tn, t, b)          ▷ decompose task into method's sub tasks
33:         stp.decompose(t, m.subTasks) ▷ impose temporal constraints on sub tasks
34:
35:         if stp.PC() = true then                 ▷ ensure temporal consistency
36:             HTN(kb, tn)              ▷ recursively refine remaining task network
37:         else
38:             return false
39:         end if
40:     end if
41: end function
```

Figure 2-11. Recovery Planning based on HTN Task Network

Figure 2-11 shows the algorithmic description of the planning procedures. The operation of this algorithm is based on the input concerning the current state of the observed service and predefined goals, which the elaborated plan solution tries to reach. Based on those inputs, the planning algorithm recursively searches and decomposes the compound tasks into non-primitive ones until all involved tasks are executable operators. The decomposition operations are conducted by searching for proper decomposition methods in the knowledge base. The line 15 -17 selects the possible refinement operation and performs the variable binding operations of the selected method. If the selected contains only operators, it gets the effects of the selected operators and update the task network by deleting the accomplished task from the current networks and insert the operator as part of solution plan. The line 25 to 29 uses temporal critics to validate and ensure the temporal consistency of the current partial plan. If the selected task contains further decomposition method, it will be further decomposed in to sub-tasks, with which the target task network will be extended with new sub-tasks. The procedure is operated recursively until all tasks in the network are non-decomposable.

2.4.4 Integration of Temporal Reasoning and HTN Planning

For the temporal constraints, we use an STP (simple-temporal-problem) data structure to catch not only qualitative ordering constraints, but also quantitative temporal relations. As the recovery plan is being defined by the HTN algorithm, temporal constrains are continuously added to the STP data. Every plan π has an associated STP with start and end time points, T_s and T_f and for every operator $o_i \in \pi$ two additional time points, s_i and f_i, representing the start and the end time points of the respective operator. Every time a task is decomposed by a method, temporal constraints between its sub tasks and the remaining tasks are posted to enforce the ordering structure of the sub workflow as encoded in the decomposition methods. Whenever a task is accomplished by an operator, the task's temporal properties are inherited by the operator. A durative action *ai* with duration *d* is encoded by the constraint [*d, d*] between the time points s_i and f_i, a task τ with a deadline *t* generates the constraint [0*, t*] between the time points T_s and τ_s. Algorithm 3 details the *decomposeTo()* algorithm used to pass temporal properties from tasks to sub planning elements. Figure 2-12 shows the decomposition method.

```
function DECOMPOSETO(PlanningElement p, PlanningElement[] into, OrderingConstraint[] c)

    for all PlanningElement e ∈ into do
        stp.add($e_s$, $e_f$)                          ▷ add start and end time points
        stp.addConstraint( ($p_s \rightarrow e_s$) )     ▷ e must start after the decomposed task p
        stp.addConstraint( ($e_f \rightarrow p_f$) )     ▷ e must end before the decomposed task p

    for all OrderingConstraint ($p^1 \rightarrow p^2$) ∈ c do
        stp.addConstraint( ($p^1_f \rightarrow p^2_s$) )   ▷ add remaining ordering constraints
```

Figure 2-12. Decomposition Method

2.4.5 Plan Validation with Policy Engine

The policy engine can be integrated into the HTN planning algorithm. Using the *isVaild()* interface: *boolean isVaild(KnowledgeBase kb, Plan π),* the planner inquiries the policy engine in every planning step, if the intermediate recovery plan and the preliminary knowledge base state comply with policies from the policy repository. Failure in the validation of such policies causes the planner to backtrack. The policy engine is not yet implemented in the current planning system; however, to build an extension in the planner to enable such capability is relative straightforward.

2.5 Policy-driven Dynamic Network Configuration

An effective way to optimize the usage of network resources is to control traffic routing and subsequently support Quality of Service (QoS). By specifying the manner in which traffic within a given network should be routed, the performance of supported services can be optimized by balancing the load distribution or minimizing the bandwidth consumption in the network. Also, in the case of link failures, alternative routes can calculated and configured so that the impact on affected services can be minimized.

This work focuses on IP-based networks whereby each network link is assigned a link weight and traffic flows are routed along shortest paths to destinations. The shortest path is defined as the route between two nodes with the least total sum of link weights. Traffic routing can be controlled by setting appropriate link weights in the network by taking into account the overall traffic demand, so as to satisfy high-level objectives such as improving load balancing while achieving acceptable QoS. The network is also assumed to be supporting Differentiated Services (*DiffServ*) [21] whereby several traffic classes may exist and each requires different treatment. Within this framework, critical information can be prioritized and treated differently.

In general, it is hard to find an efficient algorithm for achieving an optimal solution for the link weight setting problem. We therefore design and implement a heuristic algorithm based on the Tabu search method for solving the problem. The algorithm virtualizes the network into separate network planes so that traffic of different priorities can be forwarded with different QoS and produces a set of link weights for each virtual network plane.

This section presents an approach based on the policy paradigm for dynamically (re-) configuring IP networks so that the desired quality is provided to supported services. The feasibility of the approach is demonstrated with two case studies applying to military networks.

2.5.1 System Architecture

As illustrated in Figure 2-13, the proposed management system is composed of two subsystems: *Policy-based Management (PBM)* and *Network Dimensioning (ND)*. Network-level policies are entered in the PBM subsystem, stored in a repository and subsequently enforced, influencing the functionality of the ND subsystem in order to address the high-level objectives of network administrators. Based on the requirements specified in the policies, the ND subsystem then optimizes the network by executing the optimization algorithm.

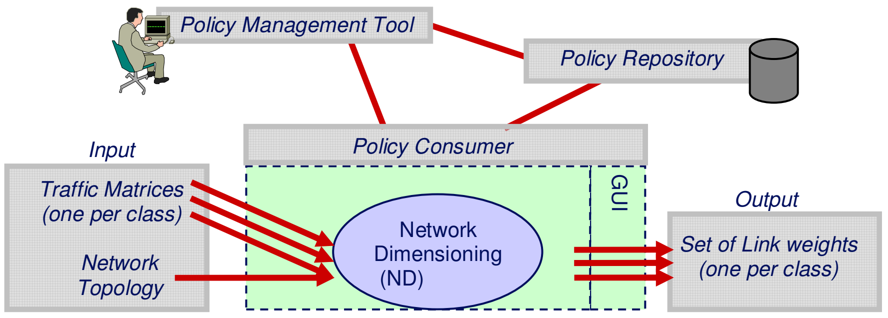

Policy Management Tool
Policy Repository
Policy Consumer
Input
Traffic Matrices (one per class)
Network Topology
Network Dimensioning (ND)
GUI
Output
Set of Link weights (one per class)

Figure 2-13. Overview of the policy-driven management system

A Java-based graphical interface is also implemented to provide a clear and controllable representation of the outcome produced by the ND subsystem. Users can view how each traffic class is being routed across any source/destination pair and the load of each link before and after the optimization. Statistical information given in plots can be instantly generated to facilitate users in understanding the effect of the optimization.

2.5.1.1 Design of Network Dimensioning Components

The core of the ND subsystem is the link weight optimization algorithm which incorporates the multi-topology (MT) concept for handling multi-traffic class scenario. It provides a means to configure class-based routing for different types of traffic. The physical network is virtualized as separate network planes (or virtual topologies). Each traffic class uses a specific routing table for that virtual topology. In our algorithm, MT is used to isolate the routing of each DiffServ Per-Hop Behavior (PHB) by providing different link weight settings for individual PHBs. Hence, packets of different PHBs can be routed independently from one another. This advanced feature is supported by configuring multi-topology protocols such as M-OSPF [22]. Coupled with the link weight optimization algorithm, an optimal solution can thus be computed for the multi-QoS class scenario.

Figure 2-14 illustrates the concept of MT-TE and serves as an example of how MT provides intra-domain path diversity across three virtual topologies between a single source/destination pair. With default link weight as 1 for all links, all traffic flows are routed via the middle path of the topology, causing congestion. However, using MT-TE, our ND subsystem can compute a set of dedicated link weights for each traffic class. Based on individual link weight settings, traffic flows of each class may follow a different path. Congestion in the middle path is thus alleviated.

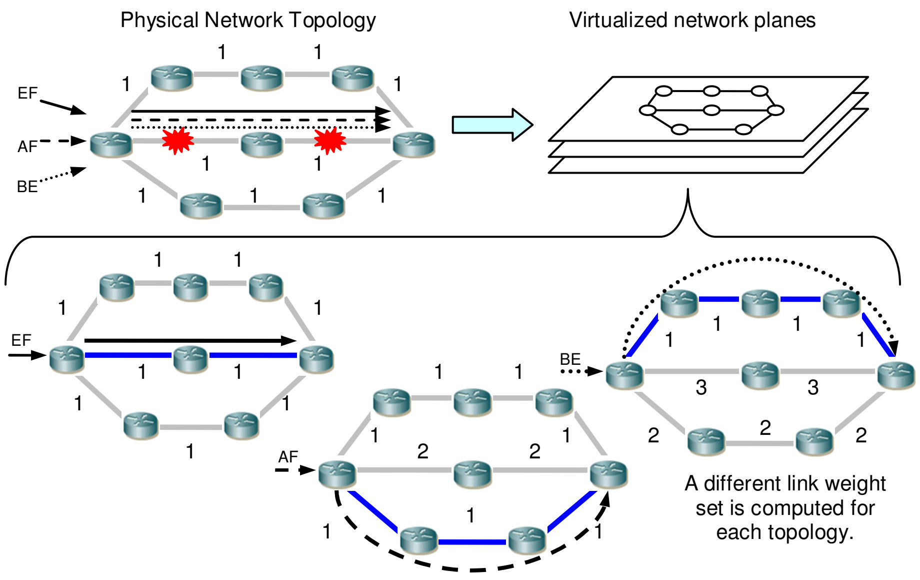

Physical Network Topology
Virtualized network planes
EF
AF
BE
A different link weight set is computed for each topology.

Figure 2-14. MT-TE: The physical network virtualized as separate logical topologies with each having a different link weight setting. Congestion at the shortest path (middle route) is avoided

The behavior of the ND algorithm can be directly influenced by the policy directives generated by the PBM subsystem. For instance, the optimization algorithm may use a different optimization objective for achieving the desired performance represented by different policies. Besides the input from the PBM subsystem, the algorithm also requires two other inputs: the network topology (including link capacities) and the expected traffic demand (in the form of estimated traffic matrices for each traffic class).

2.5.1.2 *Design of Policy Components*

Policy-based management [23] provides the ability to (re-)configure networks so that desired QoS goals are achieved. The approach facilitates flexibility and adaptability as policies can be dynamically changed without modifying the underlying implementation. The key components of Figure 2-13 are briefly described below.

- *Policy Management Tool (PMT)*: The PMT provides the policy creation environment through which a network administrator can enter new policies. The latter are of the form *if <condition> then <action>*, where the conditional part can be a compound expression encapsulating network state and events. The *<action>* expression can be a set of actions that specify the way in which the optimization algorithm should run to achieve the high-level objectives.
- *Policy Repository (PR)*: The PR is a centralized component based on an Lightweight Directory Access Protocol (LDAP) implementation that stores policies after they have been translated into object-oriented representation. Once a new policy is stored, activation information is passed to the Policy Consumer in order to retrieve and enforce it when the relevant conditions are met.
- *Policy Consumer (PC)*: The PC is responsible for enforcing policies on the fly while the network is operating. Before enforcing a policy, the consumer communicates with the PR and downloads the relevant policy objects. These are subsequently used to generate a script that implements the policy, which is interpreted into management operations when the policy is enforced. The latter can be achieved either statically through the PMT, or dynamically based on network events, and in both occasions management operations involve setting optimization attributes of the ND algorithm.

2.5.2 A Heuristic Multi-QoS Class Link Weight Optimization Algorithm

A network is modeled as a directed graph $G=(V,A)$ where V and A represent the set of nodes and links respectively. Each link $a \in A$ has a capacity denoted by $c(a)$. We have a traffic matrix D that for each pair $(s,t) \in V \times V$ represents the demand $D(s,t)$ in traffic flow between source node s and destination node t. With each pair of (s,t) and each link a, we associate a variable $f_a^{(s,t)}$ telling how much of the traffic flow from s to t goes over a. Variable $l(a)$ represents the total load on link a, i.e. the sum of the flows going over a. Furthermore, we denote the utilization of link a by $u(a) = l(a)/c(a)$.

With the above notation, the problem can be formulated as below:

$$Minimize \ \ \Phi \tag{2.6}$$

subject to:

$$\sum_{x:(x,y)\in A} f_{(x,y)}^{(s,t)} - \sum_{x:(y,z)\in A} f_{(y,z)}^{(s,t)} = \begin{cases} -D(s,t) & \text{if } y=s, \\ D(s,t) & \text{if } y=t, \\ 0 & \text{otherwise,} \end{cases} \tag{2.7}$$

$$f_a^{(s,t)} \geq 0 \qquad \text{a} \in \text{A; s,t} \in \text{V} \tag{2.8}$$

Constraints (2.7) are flow conservation constraints that ensure the desired traffic flow is routed from s to t.

In our policy-based management system, we implemented three different ND optimization objectives for policies to be mapped to.

Optimization Objective 1 (OBJ-1): Minimize the maximum link utilization:

$$\Phi = \operatorname*{Max}_{\forall a \in A} \mathrm{u(a)} \tag{2.9}$$

Maximum link utilization is defined as the highest utilization among all the links in the network. Minimizing this value ensures that traffic is moved away from congested to less utilized links and is balanced over the links.

Optimization Objective 2 (OBJ-2): Minimize the average link utilization:

$$\Phi = \frac{\sum_{\forall a \in A} u(a)}{|\, A \,|} \tag{2.10}$$

The goal of this optimization objective is to minimize the overall network link utilization. It tries to minimize the total bandwidth consumption in the network by shortening the routes to be used for traffic delivery.

Optimization Objective 3 (OBJ-3): Minimize the weighted link utilization:

$$\Phi = w \cdot \operatorname*{Max}_{\forall a \in A} \mathrm{u(a)} \; + (1-w)\frac{\sum_{\forall a \in A} u(a)}{|\, A \,|} \tag{2.11}$$

This optimization objective minimizes the weighted sum of the maximum and average link utilization. It allows policy users to specify the value of w in order to adjust the importance and balance of the two optimization objectives.

We propose a Neighborhood Search Algorithm (NSA) based on the Tabu Search technique for solving the problem efficiently with respect to performance and computation complexity. The NSA is an important tool to solve hard combinatorial optimization problems efficiently. The basic steps of NSA can be summarized as follows. Consider a starting solution x. NSA explores the solution space by identifying the neighborhood of x, $N(x)$. The neighbors of x are solutions that can be obtained by applying a single local transformation (or a move) on x. The best solution in the neighborhood is selected as the new current solution. This neighborhood searching iterates until the stopping criterion is satisfied. The algorithm returns the best visited solution.

During neighborhood search, NSA can move the current solution to the best neighbor that either improves or worsens the quality of the solution. To avoid cycling, a special memory list is used to store previously visited solutions for a certain number of iterations. A neighbor solution is rejected if it is already in the list. To increase effectiveness, an intensification or diversification technique is used to force the algorithm to explore parts of the solution space that have not been searched yet.

2.5.3 System Evaluation and Analysis

This section illustrates how the proposed policy-driven management system can be applied to practical military networks. For the case studies presented here, we take a sample military network topology of 14 nodes and estimated traffic matrices.

2.5.3.1 Case Study 1

A military helicopter carrying an important person crashes at a politically sensitive site. The exact coordinates of the site are unknown. A rescue mission to recover the personnel is initiated. An unmanned surveillance vehicle that provides a video feed back to the headquarters is deployed to determine the crash site. Meanwhile, a link within the network fails causing the quality of the video to deteriorate significantly because of congestion. The quality of the video service must be immediately restored. The following policy is applied to the network with the new information of the failed link.

***if** event(rescueOp) && event(congestion) **then** optimize(OBJ-1)*

This policy is triggered by the run-time event of network congestion during a rescue operation. Since priority in such missions should be given to supporting services (e.g. video feed from the rescue location), the policy will result to a congestion-resolving configuration, despite the fact that other lower-priority services may suffer longer delays.

We show in Figure 2-15 screenshots from our system. Note that the failed link (i.e. link 4-10) has already been removed from the network topology. Before applying the policy we can clearly see that link 3-4 and 6-10 are overloaded (i.e. traffic flowing through these links suffers QoS degradation). After the optimization, these links are no longer overloaded while several links carry increased traffic (e.g. link 3-8 and 10-11). Some traffic flows have been diverted to other routes to alleviate congestion.

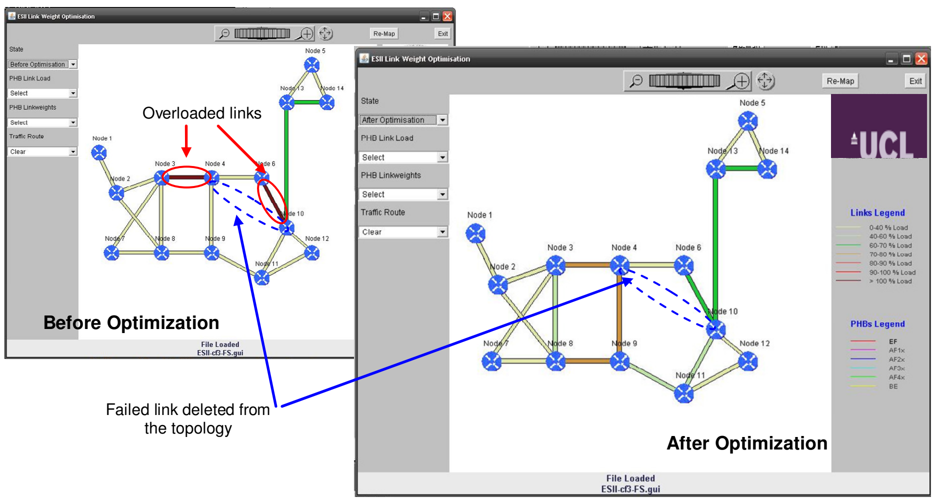


***Figure 2-15. Re-optimizing* the network to sustain QoS after link failure.**

2.5.3.2 Case Study 2

New intelligence gathered indicates possible attacks. The network is to be set to battle readiness mode (i.e. preparing to accommodate new missions and thus having increased network load). This requires the network to have minimal average load without causing QoS degradation to in-progress services. With the weight w of *optimization objective 3* initially set to 1.0, the following policy is applied to the network, causing dynamic re-configuration:

***if** event(battleReadiness) **&&** maxLinkLoad < 90% **then** decrWeight(20%)*

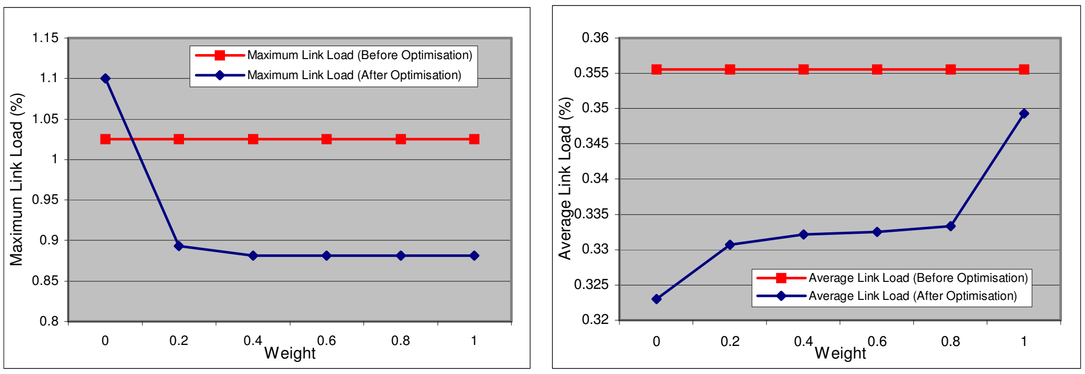


***Figure 2-16. Optimization* with different weights: (left) maximum link load,(right) average link load**

We present the results of the optimization via *OBJ-3* by varying the weight from 0.0 to 1.0 in Figure 2-16. The plots show how the weight can tune the optimization results from one objective to another. When the weight is 0.0, we are essentially optimizing the network via *OBJ-2*. Conversely, when the weight to 1.0, we are actually using *OBJ-1* as the ND optimization objective.

In this case study, the objective is to prepare the network for new incoming mission traffic. At the same time, in-progress services are not to be disrupted by congestion. Hence, the policy runs the ND algorithm iteratively with decreasing weight until the maximum link utilization exceeds 90%. From our results, we can see that the optimal weight setting is 0.2 where the average network is decreased without having any link higher than 90% utilization.

2.6 Conclusions

This section presented an HTN-based planning algorithm designed to assist the planning operations for IT service fault recovery. The planning process is based on the refinement of high-level recovery planning tasks into executable operators, which could be implemented directly on the system to achieve the designated recovery objectives.

Since the planning operation in a complex operational environment is a knowledge intensive task, we propose a recovery knowledge base component to facilitate the planning operation. It provides the planner with a unified access to the supporting information for the planning procedure. To represent the recovery knowledge, we propose several information models, which encode recovery methods, operators as well as constraints and task networks. The presented information model is in alignment with the HTN based planning paradigm and can be directly used in the proposed planning algorithm. To integrate more management information into the planning process, the knowledge base could be extended with interfaces to other management systems so as to enrich the fault recovery knowledge. For example, the distributed knowledge discovery component discussed in the previous section could be connected with the knowledge base in order to enrich and update the existing information items.

In real world fault recovery scenarios, the operations are usually constrained by different factors, such as temporal condition, precedent of actions in the sequences. To reflect this aspect, we integrated a temporal reasoner to show how such constraints could be integrated into the planning process.

This section also described a policy-based management system with a user-friendly graphical interface for optimizing the performance of services in IP-based networks. The system is driven by policies that optimize the network for different objectives. It is also capable of dealing with conflicting optimization objectives via a weight acting as a tuning parameter. A link weight optimization algorithm employing the Tabu search method was developed to influence the dynamics of routing traffic within the network. The concept of multiple virtual topologies was incorporated in the algorithm for handling multi-QoS class scenarios. The results suggest that the pure IP-based DiffServ network can be optimized for various objectives through the intelligent assignment of link weights.

3 Federated Management in the Future Internet (FeMaFI2)

3.1 Introduction

In an environment where different autonomous entities will be sharing the same set of resources, federation mechanisms need to be in place such the resources are used effectively towards specific objectives. FeMaFI2 builds upon the architecture developed in the previous phase of the project that supports governance and negotiation functionalities. The objectives of this activity for the year 2009 were established as follows:

- Specification of policies for the orchestration of Autonomic Management Systems (AMSs).
- Design of algorithms and methodologies for the orchestration of AMSs based on the defined policies.
- Definition of mechanisms and protocols to establish negotiation between multiple service providers in competitive environments.
- Enhancement of the negotiation approach with automated resource negotiation models based on game theoretic approaches.

This section concentrates on the first, second and forth objectives[1] because the third one was already covered in Deliverable D9.4 issued in March 2009.

3.2 The Orchestration Plane Concept

The purpose of the Orchestration Plane (OP) is to govern and integrate the behaviours of the network in response to changing context and in accordance with applicable high-level goals and policies. It supervises and integrates all other planes behaviour insuring integrity of the Future Internet management operations. The OP can be seen as a control framework into which any number of components can be plugged into or out in order to achieve the required functionality.

The OP would also supervise the optimization and the distribution of knowledge within the Knowledge Plane to ensure that the required knowledge is available in the proper place at the proper time. This implies that the OP may use either very local knowledge to deserve a real time control as well as a more global knowledge to manage some long-term processes like planning.

The OP would host several Autonomic Management Systems (AMSs). It is made up of one or more Distributed Orchestration Components (DOCs)[2], and a dynamic knowledge base consisting of a set of data models and ontologies and appropriate mapping logic and buses. A DOC, which controls a single orchestration domain, enables the AMSs of the domains to communicate and cooperate with each other. Orchestration domains coincide with the administrative domains (resources under the same administrative

[1] The background research supporting this section of FeMaFI2 has been done within the Autonomic Internet (AutoI) research project where both UPC and UCL participate

[2] The term "Distributed" in the denomination of a DOC is to emphasize the fact that the constituting parts of a DOC are distributed software components

responsibility) and, as a consequence, the set of orchestration policies within the orchestration domain is homogeneous.

Each AMS represents a set of virtual entities, which manage a set of virtual devices, sub-networks, or networks using a common set of policies and knowledge. The set of virtual resources managed by each AMS are non-overlapping. As a consequence, the responsible AMS will forcibly implement all self-* functions for its managed resources. This assumption does not imply that each resource will be used for only one service or network. A physical resource, being divided into several virtual resources, can be used for different services or networks. Each of those virtual resources, however, will be dedicated to a single service, and as such will be managed by only one AMS. The AMSs access a knowledge base, which consists of a set of data models and ontologies.

The OP acts as control workflow for the AMSs of the orchestrating domain, ensuring bootstrapping, initialization, dynamic reconfiguration, adaptation and contextualization, optimization, organization, closing down of AMSs. The OP provides assistance for the Service Lifecycle Management, namely during the actual creation, deployment, activation, modification and in general, any operation related to the management services. The DOC enables the following functions across the OP:

Federation: Each AMS is responsible for its own set of virtual and non-virtual resources and services that it governs as a domain. The Federation enables a set of orchestration domains to be combined into a larger orchestration domain, guided by common high-level goals, while maintaining local autonomy.

Negotiation: It can take place between autonomous entities with or without human intervention. DOCs and AMSs are the entities engaged in negotiations to achieve their goals. Each AMS advertises a set of capabilities (i.e., services and/or resources) that it offers for use by other AMSs. The OP acts as an arbitrator, performing negotiation between the AMSs for the fulfillment of a specific SLA, defined by the orchestration policies, user requirements and high-level goals.

Distribution: The Orchestration Plane provides communication and control services that enable tasks to be split into parts that run concurrently on multiple AMSs within the OP. In other words, The OP can assign different duties to different AMSs

Governance: Each AMS can operate in an individual, distributed, or collaborative mode (i.e. in federation). Due to their inherent autonomic behaviours, AMSs may from time to time take local decisions that go against the goals of the orchestration domain. Thus, the DOCs monitor the actions of the AMSs, triggering responses (i.e. re-negotiation of AMS and/or DOC policies, closing down or bootstrapping of AMSs) to such violations.

System Views: The OP is responsible for managing the system views that are stored and diffused using the knowledge plane (detailed in [26], section 3.3.4). DOCs will fetch the information required for their operation from the AMSs as well as the services and resources (through the vSPI interface defined in [27], section 7).

Throughout this section the term high-level policies or business objectives are particularly linked to high-level aspects of management and control over a given system. This is, by no means, aligned to economical aspects such as pricing strategies. As described in [25], policies are conceived in the context of a *policy continuum*. In such a *continuum*, policies are refined from high-level policies into lower level policies that can be applied to a specific instance of equipment. The following levels of policies are defined:

- **Orchestration level policies –** These policies control the negotiation, distribution, governance, and federation tasks of the OP.
- **System level policies –** Those policies are related to the autonomic management systems (AMSs). System level policies are used to define the operation of the AMSs.
- **Component level policies –** Component level policies manage the network virtual resources.
- **Instance level policies –** Instance level policies are embedded inside devices that can perform their own decision functions.

The OP deals mostly with orchestration level policies. Within orchestration policies, negotiation, federation, distribution and governance policies are defined. Those control the possible actions of the DOCs on each of those tasks. DOCs deal with lower level policies only during negotiation processes. In those cases, the OP must check that the *system level policies* produced by the AMSs are aligned with the *orchestration policies*. To improve the readability of this document, from now on the term *policy* refers to *orchestration policies*. Other levels of policies (i.e. *system level policies*) will be clearly contextualized by using their full name.

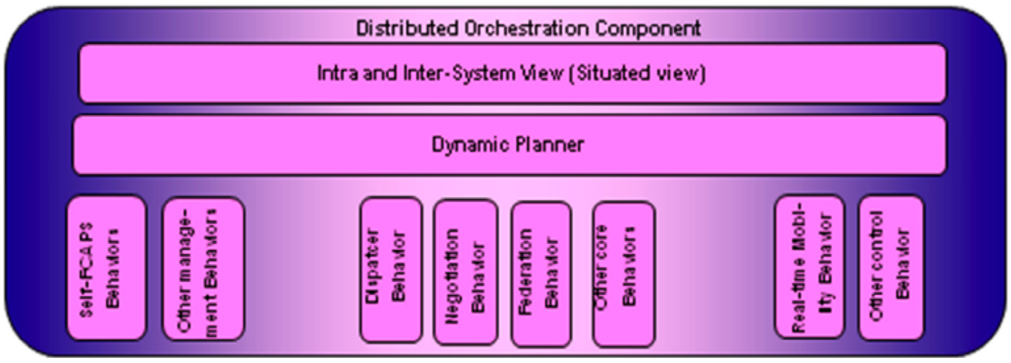


Figure 3-1. **The architecture of the Distributed Orchestration Component.**

System Architecture

The architecture of the DOC is shown in Figure 3-1. The Dynamic Planner (DP) component bootstraps and closes down AMSs and Behaviours, following a set of workflows. Its actions are dictated by the orchestration policies and the workflows. Behaviours complement the Dynamic Planner, implementing the task specific actions that are required from the OP. They use specialized knowledge of the problem at hand in order to solve it efficiently. It is up to the DP to choose the right Behaviour for each given task, controlling its lifecycle: instantiation, execution, and finalization.

Behaviours perform the specific/individual orchestration actions required to be performed by a DOC and also represent specific management tasks on the network. The main behaviours are *distribution*, *federation*, *negotiation* and *governance*. Those Behaviours are described in detail in the following sub-sections.

DOCs use the knowledge plane described in [26] to store and disseminate the information required for their operation. This information can be decoupled into two parts, or views, according to their relevance to a given DOC. The Intra-System View concerns information required to orchestrate the services within the orchestration domain, while the Inter-System View deals with the orchestration of several orchestration domains. The Intra-System View contains information that enables DOCs to become aware of the particular situation that they are now in; the Inter-System View provides similar information for collaborating DOCs.

The following sections of this document provide an in depth description of the DP, the Behaviours and the interfaces of the DOCs. For each component of the DOCs, we describe their interaction within the OP and with other planes. We also describe their policies and comment on their scalability and stability.

3.3 Dynamic Planner

The Dynamic Planner is the central entity of the DOCs. It is responsible for dispatching Behaviours, which will take care of specific orchestration tasks. The DP relies on behaviours for the implementation of specific orchestration tasks, so its design can be as generic as possible. In addition, the DP relies on orchestration policies and workflows, defined by the operators. Those will define the actions to be performed by the DP to accomplish any orchestration tasks. To avoid unnecessary resource consumption of the networking components, the DP is an event-based component; it makes use of the monitoring facilities of the knowledge plane, as well as those of the governance behaviours, to trigger notifications of important event changes. Finally, the DP distributes policies and SLAs to the AMSs. This is performed at the deployment of new AMSs, or whenever the orchestration policies change.

The DP, being a generic execution engine of orchestration tasks (or meta-management of self-governing management systems), requires tools to describe the tasks that must be executed when certain events take place. This is performed using the workflows detailed below.

3.3.1 Orchestration Workflows

Orchestration workflows are event-based, that is, the DP acts only when events occur. Events may be triggered from within the network (i.e. a conflict has happened), or from the outside (i.e. the operator defines a new set of goals for the OP). The following types of events exist:

- **Parameters changes:** A change occurred in a set of variables, which have values within a certain range. One such event could be the delay of a link becoming too high, which may require the renegotiation of the SLAs or the redeployment of an AMS.
- **Conflicts:** A conflict within two AMSs or behaviours has been detected. Such conflicts will usually come from the governance behaviours.
- **Federation requests:** The operator, or an AMS, has requested a federation action.
- **Separation of a federation requests:** The operator, or an AMS, has requested the separation of two networks.
- **Distribution requests:** The operator, or an AMS, has requested that a new AMS to be deployed.
- **Request to close down a component:** The operator, or an AMS, has requested the closing down of an AMS.

- **New goals and high-level policies:** The operator has defined new goals, policies or user requirements that must be satisfied by the DOC.

Those events trigger workflows, which are described in a domain-specific language (DSL) [25]. The language uses an orchestration vocabulary to simplify the programming of workflows. The idea of this meta-language is to offload most of the functionalities to the Behaviours or to the AMSs, so the DP stays simple and fast. Thus, the vocabulary of this DSL is quite small, and the DP relies on the implementation of specific behaviours to perform the bulk of the orchestration tasks. For example, if there is a need to solve a specific problem, the DP will identify that such a problem occurred by monitoring certain variables within a set of pre-defined boundaries, and then it will trigger a Behaviour that is suitable for that situation.

3.3.2 Dynamic Planner Policies

The DP uses policies to set limits on the execution of the workflows defined by the operator. Those policies define the maximum amount of resources (CPU, memory, time, number of running workflows) used for each of the workflows. Orchestration policies also define the action taken by the DP when a policy fails or when a certain orchestration event is not captured by any workflow. The DP policies also provide means to improve the evolvability of the installed workflows and policies. By means of default policies, operators are able to identify situations that were not foreseen at the design of the workflows and policies. Further, policies are one of the tools to ensure the stability of the DP.

Resource limiting policies: resource limitation policies at the DP level in essence assure the stability of the system. They are put in place to terminate ill-behaved workflows, which in our definition are workflows using too much resources or taking too much time to complete. Those policies are defined on a per-workflow basis, as well as a generic policy for all the workflows. In case a certain workflow does not have an associated resource-limiting policy, the default limiting policy is used if it exists. Being the centralized entity that controls the bootstrap and closing down of Behaviours, the DP also defines resource limitations to Behaviours by means of specific policies. Those policies have the same properties and functions of the policies described previously for workflows.

“Multi-tasking” policies: they limit the amount of multi-tasking of each workflow, that is, it constraints the amount of workflows that may be triggered by each workflow. A DP-wide policy can also be defined, in order to specify the maximum amount of workflows running at each DP. Excess workflows may be put in an execution queue, or simply rejected. Similar to the resource limiting policies, multi-tasking policies are defined to improve the stability of the DP platform. They may catch ill-behaved workflows or design flaws in the writing of the workflows. Again, those policies are either defined for each specific workflow, or for the all the workflows of the DP, allowing certain workflows to create workflows in excess of the other workflows’ default limits.

Default action policies: those policies define actions to perform when a workflow fails, or when no workflows are attached to a certain event. This approach is similar to the *default* action of *switch* clauses in programming languages, which is used to catch unpredicted situations or to handle errors. One of the uses of default action policies is to communicate to the operator that the DOC found a situation that was not contemplated by the installed workflows. As an example, when an ill-behaved workflow overflows its allocated resources, the operator could be notified, receiving the name of the workflow as well as some measurements related to the state of the network and the state of AMSs and Behaviours. Similarly, in some situations the installed workflows may not deal with a certain event. In case this happens, a default action is taken. This action may consist of notifying the operator, as well as shutting down or restarting the problematic AMSs.

3.4 Behaviours

Behaviours are the specialized components of the OP that are responsible for implementing the orchestration tasks. Due to the specificity of the Behaviours, the OP is able to cope with different management and virtualization technologies. Behaviours are essentially software components that are executed by the Dynamic Planner on a per-case basis. The Dynamic Planner starts up the appropriate Behaviours, as defined by its workflows, passing them the parameters that they need to perform their function. As an example, the DOCs could implement Behaviours for the negotiation of high-level policies, the distribution of tasks, the creation and destruction of services and virtual routers. Behaviours interact with each other when necessary, i.e. the Federation Behaviour may interact with QoS-related Behaviours if the required QoS couldn't be met when two networks are joined.

This section details the most important behaviours, which implement the governance, distribution, negotiation, and federation tasks that are carried out by the OP. Those are called the **core behaviours**, since they will be required for the operation of any autonomic network based on our approach. Other types of behaviours can also be defined for an improved management of specific cases. For example, a Mobility Management Behaviour, an Accounting Behaviour, among others, could be deployed as well.

Finally, Knowledge Update behaviours may be defined. Those behaviours are responsible for updating the intra- and inter-system view that is needed for the core behaviours or for the specialized behaviours. Knowledge Update Behaviours define the "*What, When* and *Where*" of the information (what information to collect, when to collect, and from whom). Those Behaviours are specific to each service. However the whole set of these Behaviours supervises storage of information in the Knowledge Plane.

In order to control the behaviours, a set of standardized functions must be implemented. Those control the lifecycle of the behaviour (start up, closing down) and its parameters (policies and goals). Specific behaviours (i.e. a negotiation behaviour, a distribution behaviour) may specify more functions that may be called by the DP or by external components. In order to communicate with the Dynamic Planner, all Behaviours implement a set of basic primitives. Those primitives were derived from the basic functions that most behaviours do, based on [25], [27] and [28]. This allows the DP to perform a set of basic operations on the Behaviour, such as starting up and closing down and changing their policies.

3.4.1 Distribution Behaviour

The distribution behaviour provides communication and control services that enable management tasks to be split into parts that run on multiple AMSs within an OP. The distribution behaviour, thus, controls the deployment of AMSs and their components. Using orchestration policies and based on user requirements, this behaviour will deploy the policies and SLAs driving each AMS, as well as which virtual resources they will control (in the case of the deployment of multiple AMSs). It is up to the SP and the VP to define on which virtual resources the AMSs will take control.

Further, the distribution behaviour only specifies the characteristics of AMS deployment. It is up to the Service enablers Plane (SP) to perform the deployment. As such, the distribution behaviour uses the service deployment interfaces.

3.4.1.1 Distribution Policies

Policies in the distribution behaviour define the capabilities that a given AMS must exhibit. Further, distribution policies define the QoS constraints and SLAs to be enforced by the AMSs. Some of those parameters may be negotiable, while others may not. This information is also coded as policies.

Capabilities policies: Those policies describe, in a somewhat abstract way, the capabilities that the deployed AMSs must have. They may define the type of autonomic management that it should perform (i.e. self-configuration, self-optimization), as well as the service that it will manage (multimedia, Web, telephony). Those requirements will usually be forwarded to the ANPI interface, so the SP will return a list of the registered AMSs that may provide the requested capabilities. Note that, due to the self-governance of the AMS, the distribution behaviour only defines in a high level the type of service that must be managed. It is up to the AMS to carry out this management.

QoS and SLA-related policies: Those policies will guide the AMSs in the set up of appropriate mechanisms for QoS management of given services. The QoS and SLA policies are derived from the high level goals defined by the operators of the orchestration domains and of the federation that the DOC belongs to. The policies define ranges of values for different QoS parameters, and if those parameters are changeable or not. This allows the AMSs to redefine the QoS constraints of specific components of the service as well as the entire service. Other parameters cannot be changed, since values outside this range would go against the high level goals of the orchestration domain. If any of those static QoS guarantees cannot be provided by the AMS being deployed, the AMS warns the DOC. The DOC then starts up the negotiation behaviour in order to derive a new set of high-level goals.

3.4.2 Negotiation Behaviour

In Future Internet networks, it is necessary to offer and publish their supported services so that more complex services can be provided. For flexibility purposes the creation and establishment of complex services must be done in an autonomic manner. The deployment of complex services usually involves the collaboration of multiple entities having different capabilities, controlling the resources under its domain, and in summary, providing a number of network and application services. An important component that allows this requirement to be met is the negotiation component of the OP. This component acts as a virtual service broker that mediates between different AMSs, taking care of service requests and providing support so that the underlying service providers can negotiate SLAs, responsibilities, tasks, high-level goals, etc. In order to support such functionality, the negotiation behaviour takes into account the nature of the underlying service providers, the services they provide, their interests, their service qualities and other key aspects. All in all, this functionality should be fully automated with complex negotiation-oriented decision-making processes.

In our conceived management framework each AMS advertises a set of capabilities (i.e., services and/or resources) that it offers for use to the OP. The negotiation functionality enables AMSs to reach agreements and form SLAs for selected, well-described services. DOCs mediate during the negotiation process acting as trusted third parties or service brokers for the AMSs since the DOCs have the advantage of a more holistic view of the network. Examples include using a particular capability from a range of capabilities (e.g., a particular encryption strength when multiple strengths are offered), being granted exclusive use of a particular service when multiple AMSs are competing for the sole use of that service, and agreeing on a particular protocol, resource, and/or service to use.

The negotiation functionality orchestrated between AMSs and DOCs has inherent business and technical concerns. The following elaborates on these two critical aspects.

3.4.2.1 Negotiation Policies

The negotiation behaviour uses policies to represent the parameters of the negotiation process, as well as the ranges of AMS and service parameters being negotiated. Policies provide a higher amount of flexibility to the negotiation behaviour, allowing the AMSs as well as the DOCs to adapt the negotiation to each particular scenario.

Negotiable-set policies: those policies are produced by the AMSs as well as DOCs, and define two different sets of policies. One set defines the non-negotiable terms of the SLA, that is, the requirements that one of the AMSs or the DOC are not willing to compromise on. The second set of policies, define which define the negotiable service parameters.

Range negotiation policies: for the negotiable policies, AMSs and DOCs may define a set of ranges or allowed values. Those policies can be used to describe the capabilities of the AMSs and DOCs, as well as what resources and service levels each of the components are willing to provide.

Orchestration-related policies: those policies act over the global negotiation process, setting values for some parameters relevant to the negotiation process. For instance, these policies could define the maximum amount of negotiation iterations, the maximum time per iteration, among others. The most important use of those policies is to define limits on the negotiation process, i.e. to improve the scalability and stability of the OP. Those policies may also define goals that must be achieved at each negotiation round, and actions if those goals are not met. For example, an AMS should provide a different, more flexible set of constraints at each round. Another policy could state that, If an AMS is inflexible after a certain number of rounds, the negotiation process is stopped.

3.4.3 Governance Behaviour

AMSs are self-governing elements, having each their own control loops over the virtual resources under their management scope. Hence, they may decide to change their policies, SLAs and user requirements on the fly, without cooperation or interventions of the DOCs. The DOCs, being responsible to oversee the operation of the AMSs in order to allow a smooth intra-domain and inter-domain operation, must hence monitor the actions of the AMSs. The Governance Behaviour performs this monitoring. This behaviour ensures that the AMSs are aligned with the high-level orchestration goals set by each orchestration domain. In a nutshell, the functions of the governance behaviour are as follows:

- **Monitoring of the actions of the AMSs**: monitoring is performed to check if the actions of the AMSs are consistent with what the DOC has requested. Hence, the DOCs watch the configuration and policies of the AMSs, always verifying if this configuration is still aligned with the goals set at the orchestration level.

- **Enforcement of policies and SLAs defined by the DOCs:** the governance behaviour checks for misbehaviours, caused by AMSs changing their system level policies. Once they happen, the DOCs will take measures to indicate to the AMS of its non-compliance. Those requests take the form of function calls requiring the AMS to use a certain set of orchestration policies or a configuration that has previously been agreed.
- **Trigger for federation, negotiation and distribution tasks upon non-compliance**: once the governance behaviour detects that an action from the part of an AMS that seems to be conflicting with the objectives of the OP, it will start up the proper counter-measures. This trigger occurs after the DOC tries to enforce a certain configuration over the AMSs, however the AMS rejects the proposed configuration. Hence, the DOC must find new ways to ensure the smooth operation of the network. Those may be achieved by the renegotiation of the system level policies of one or more AMSs, the replacement of certain AMSs by another implementation, or the need to merge/split the network.

Each instance of the governance behaviour monitors one AMS. The governance behaviour assumes that the set of virtual resources managed by each AMS are non-overlapping. As a consequence, the responsible AMS will forcibly implement all self-* functions for its managed resources. This assumption does not imply that each resource will be used for only one service or network. A physical resource, being divided into several virtual resources, can be used for different services or networks. Each of those virtual resources, however, will be dedicated to a single service, and as such will be managed by only one AMS.

3.4.3.1 Negotiation Policies

Governance policies define a set of operational parameters for the DOCs, as well as the actions that should be taken for certain violations. Thus, governance policies define the monitoring patterns that the governance behaviour must enforce. Different kinds of parameter violations will trigger different reactions. As a result, certain conditions may trigger the redefinition of policies, while other conditions may require the redeployment of the AMS. Governance behaviours are derived from the orchestration policies, high-level goals and user requirements of the services.

Due to the characteristics defined above, governance policies resemble events, as they are defined in the following form: *if one of a set of parameters lies outside the defined ranges, then do the specified actions*. The monitored parameters may comprise properties of the AMS, user-facing services or virtual links. For example, if a certain service meets a significant degradation in the QoS, the governance behaviour may decide to renegotiate the SLAs of the self-configuration AMS managing this service. The actions that are defined in the policies may involve the following operations: triggering a negotiation among AMSs, sending a message for the AMS that it is not compliant to the set of defined orchestration goals, or shutting down the AMS and creating another one.

Since the governance behaviour monitors only one AMS, there are no “default governance policies”, as in the dynamic planner policies. The DOC, however, may install the same set of standard governance policies in all governance behaviours.

3.4.4 Federation Behaviour

The federation behaviour of the DOC is responsible for the federation of networks under different orchestration domains. Since each domain may have different SLAs and policies, a federation attempt may trigger a negotiation for new policies, or even the re-deployment of services in the case that the new policies and high level goals of the network are not compatible with some of the deployed services.

3.4.4.1 Federation Policies

Policies in the federation Behaviour define the AMSs that are involved in the federation and also policies to communicate and negotiate with federation partners for the purpose of federation:

List of involved AMSs: a part of policies in the Federation Behaviour are responsible for defining which AMSs should be involved in the federation process, and then how to communicate and negotiate with them. The list of involved AMSs is based on the goal/s to be achieved by such federation and the specific requirements of what has triggered the federation. This is an important type of policies in federation, as it avoids overloading network with unnecessary control traffic from non-concerned AMSs.

Negotiable and non-negotiable policies: this part of Federation Behaviour policies defines the applicable policies in the federation process, outlining which policies are hard and non-negotiable from the point of view of orchestration, and policies which could be negotiated regarding the agreement of the involved AMSs. Based on these policies, self-governed AMSs check if the high-level federation policies are not in contradiction with their own non-negotiable policies. If there is a conflict, the AMS rejects the federation, as there is no possibility of negotiation. In contrast, if there is a disagreement in negotiable policies, the federation process continues after one or more negotiation rounds. If, after a limited number of negotiation attempts, DOCs and AMSs find an agreement on a given set of policies, the federation continues. If not, it will be rejected.

3.5 A Negotiation Framework for the Orchestration of Future Internet Networks

Future Internet Networks (FINs) are envisioned to be highly flexible and dynamic in the creation and establishment of complex services usually requiring the collaboration of multiple interested entities. Research towards the Future Internet has received a lot of attention recently [29][30][31]. The expected flexibility of FINs, although highly sought after, also gives rise to many challenges. One of them is the problem of achieving a harmonized federation (e.g. sharing / allocation of resources, distribution of tasks for fulfilling common goals) of different autonomic management systems (AMSs). These AMSs are heterogeneous by nature and may employ different subsystems and protocols while also exhibiting different capabilities or resources.

The highly flexible nature of FINs increases network volatility. With a large amount of potential interactions and expected collaboration among various relatively dynamic networks on different issues, it is clear that there is a need to coordinate these interactions so that the establishment and cancellation of different agreements among various parties is done smoothly, with all parties benefiting from the collaboration. As such, an *orchestration plane* (OP) is envisaged that will harmonize relevant collaborations without being intrusive i.e. it can coordinate activities but will not act on behalf of any of the participating entities [29]. Under the coordination of the OP, AMSs will attempt to negotiate with each other on specific issues so as to maximise their own benefit. There may be cases where a completed negotiation is revised due to detected conflicts.

For the negotiation process a game theoretic approach is proposed. Game theory has found its application in various domains and provides a useful formal framework for the investigation of the interactions among such entities [32]. There have already been various instances where game theory has been applied to communication and network problems. For example, [33] and [34] investigated the issues of resource allocation and pricing in broadband networks. Also in the area of resource management, [35] proposed the game theoretic concept of bargaining for fair bandwidth allocation among multimedia applications. Here, we apply bargaining theory to address the issue of negotiation between AMSs, under the coordination of the OP, to reach a mutually acceptable outcome. We formulate a problem of non-cooperative bilateral negotiation game scenario where AMSs have partially or fully conflicting goals. We then proceed to solve the game for both perfect and imperfect information scenarios.

3.5.1 Orchestration of FINs

In general, negotiation is needed when an AMS lacks resources in order to achieve a goal (e.g. when it cannot provision a new service with the quality that is requested, when a service needs to be provisioned across domains, etc). This AMS has then the need to collaborate with another AMS. In our framework, it will send a request to the OP in order to find a candidate AMS to collaborate. Based on the received request, the OP will first identify the AMSs that have the requested resources. The remaining candidates can then be chosen based on different criteria. The simplest one is a random selection where a candidate is randomly selected to take part in the negotiation. A more sensible method is a greedy selection criterion whereby all the possible candidates are ranked in decreasing order based on the type and quality of services available. In this selection approach, the top ranked candidate is selected to take part in the negotiation process.

We consider a set of N AMSs, $\mathbf{A} = \{A_1, A_2, \ldots, A_N\}$, governed by an OP. A negotiation process is started by a requesting AMS by initiating a request.

Definition 1 – Request: A request is defined by a 5-tuple $REQ = \left\langle id,\ m,\ \sigma,\ dl,\ \delta \right\rangle$ where id, m, σ, dl and δ are the unique request identification number, the type of service / resource required, the numerical quantification of the requested item, the time period for which this negotiation applies to and the numerical quantification of the benefit the AMSs receive per time period, respectively.

For the OP to function seamlessly, AMSs are required to provide up-to-date information with regards to their state. As such, each AMS periodically advertises information on its current service offerings (or services they are willing to share / negotiate) so that the OP always has enough information to select the candidates for negotiation. The advertisement sent by A_n where $n = \{1,2,\ldots,N\}$ at t^{th} period is denoted by $Ad_n(t)$. Once identified, OP will inform the selected AMS(s) with a start message, $START = \langle A_x, A_y, REQ \rangle$ where A_x is the AMS which initiated the negotiation and A_y is the AMS chosen to negotiate with A_x. The negotiation process starts after all involved parties receive the $START$ message. It is assumed that the agreed terms from the negotiation must be autonomously enforced via local actions under each corresponding AMS.

The negotiation takes place between two AMSs. We denote the AMS which initiates the negotiation as R and the AMS chosen by the OP as C where $R,C \in \mathbf{A}$. Both parties negotiate over the share of a commonly desirable entity (e.g. revenue), which decays over time, i.e. $P(t \mid dl,\sigma,\delta)$. The negotiation is modelled as a turn-based process that is taken over the time set $\mathrm{T} = \{0,1,2,\ldots\}$. The AMSs have the option of leaving the negotiation when it is their turn to respond. In practice, this option will be taken if the AMS realises that it cannot gain from this collaboration. For example, in the middle of a negotiation on the issue of sharing a batch of virtual machines, an AMS may decide not to negotiate further if it realizes that there is no possibility in gaining at least as much as what it would get if the resources under negotiation were used elsewhere. Hence, there is a threshold dictating the point where it prefers to reserve the resources rather than sharing them.

All AMSs are assumed to be rational. This implies that the AMSs attempt to maximise their utilities and behave consistently according to their preferences. To direct the negotiation towards an agreement, AMSs avoid quitting the negotiation process. By this, it effectively means that they accept an offer if the utility derived from accepting that offer and opting out are equivalent. The AMSs are also expected to honour the agreements (if reached). However, there are no long-term commitments i.e. each negotiation is independent, thus eradicating the case where an AMS may consider any prospective future activity amongst the AMSs. Finally, all AMSs are assumed to be aware of the above assumptions.

3.5.2 Negotiation Protocol

We develop a *strategic negotiation model* based on Rubinstein's model of alternating offers [36][37]. The model consists of three components: the negotiation protocol, the agents' (AMSs) utility functions and the agents' negotiation strategies.

The rules by which agents interact among themselves are defined by a negotiation protocol. This specifies what and when each agent must respond to. The negotiation ends when all involved agents agree to an offer or one of them decides to quit the process without agreement. Figure 3-2 illustrates how the negotiation protocol works.

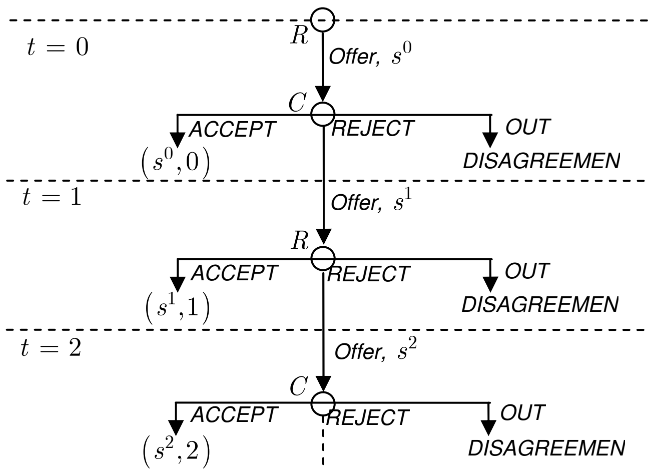


Figure 3-2. A representation of the negotiation protocol.

In general, in each period $t \in \mathrm{T}$, one agent makes an offer, s^t, as a possible resolution and the other agent responds with either ACCEPT, REJECT or opt OUT. The offer will be enforced if it is accepted, while the negotiation is considered to have broken down if any agent opts out. Finally, the responding agent who rejects the offer will have to make a counteroffer at $t+1$ to which the first agent must respond. This procedure repeats until an agent accepts or opts out of the negotiation. In this work, we assume that R always makes the first offer i.e. s_R^0.

Definition 2 – Offer: An offer by agent $i \in \{R,C\}$ at time t, $s_i^t \in \mathbf{S}$, is a pair (α_i,β_i) in which α denotes R's portion while β is C's portion of the entity under negotiation, $P(t)$. We denote $P(0) = M = \sigma \cdot \delta \cdot dl$ (i.e. M represents the total value of the entity under negotiation viewed from the beginning of the game. Note that since $\mathbf{S} = \{(\alpha_i,\beta_i) : \alpha + \beta = 1\}$, α singularly decides β.

Definition 3 – Agreement: An agreement at time t is the accepted offer and is denoted by (s^t,t). If an agent opts out at time t, then the agreement is (OUT,t).

3.5.3 Utility Functions

Each agent has its own utility function, which is used to evaluate different possible terminations of the negotiation. The agent's utility function covers all possible outcomes i.e. $U_i : \{\mathbf{S} \cup \{OUT\} \times \mathrm{T}\} \rightarrow \mathbf{R}$. Further, the agents are only interested about the final outcome of the negotiation but not the sequence of offers that leads to an agreement. In this paper, we assume that the agents use finite-horizon models with fixed costs per time unit, denoted by c_i where $i \in \{R,C\}$.

Definition 4 – Agent's utility function: Given the numerical value of M and the deadline of the negotiation and knowing its own negotiation cost, agent i's utility can be defined as $U_i(s,t) = f(s,t \mid M,dl,c_i)$. Let s_{pos} be a possible outcome of the negotiation. Then, agent i's utility at time t is given as $U_i(s_{pos},t) = U_i(s_{pos},0) \cdot \left(1 - t/dl\right) - t \cdot c_i$. For this work, we assume the agents' utility function as:

$$U_i\left(t\right) = \pi_i \sigma\delta(dl - t) - tc_i = \pi_i M(1 - t/dl) - tc_i \tag{3.1}$$

where $\pi = \alpha$ if $i = R$, and $\pi = \beta$ if $i = C$.

The utility computed from the current proposed offer is referred to as the *offered utility*.

3.5.4 Subgame Perfect Equilibrium (SPE)

At this point, we need to consider the problem of how a rational agent will choose its negotiation strategy. Usually the Nash Equilibrium [38] is used. However, the Nash equilibrium may be unstable in intermediate stages of the negotiation process. Moreover, it does not take into account the process of offer and counter-offer that characterises bargaining and the possibility of breakdown. For our case, we use the stronger concept of Subgame Perfect Equilibrium (SPE) [39]. A strategy is a SPE of a model for alternating offers if the strategy profile induced in every subgame is a Nash Equilibrium of that subgame.

We establish the preference relation of each agent. ($\succ_i$) over the set $(\mathbf{S} \times \mathrm{T}) \cup \{OUT\}$ in order to determine the equilibrium point. We assume that each agent's preference relation satisfies the following:

Assumption 1 (A1) – Opting out is the worst outcome: For every $s \in \mathbf{S}$ and $t \in \mathrm{T}$, $(s,t) \succ_i (OUT,t)$ where $i \in \{R,C\}$.

Assumption 2 (A2) – The entity under negotiation is desirable: If $r > s$, then $(r,t) \succ_i (s,t)$ where $i \in \{R,C\}$.

Assumption 3 (A3) – Negotiation time is valuable: Both agents prefer to reach an agreement as soon as possible since every round of negotiation will cost, while M decays over time. Hence, for $\forall t_1,t_2 \in \mathrm{T}$, $s \in \mathrm{S}$ and $i \in \{R,C\}$, if $t_1 < t_2$, then $(s,t_1) \succ_i (s,t_2)$. This relationship also applies to the case where the negotiation ends with an agent opting out, i.e. $(OUT,t_1) \succ_i (OUT,t_2)$.

Assumption 4 (A4) – Agreement's cost over time: For $\forall t_1,t_2 \in \mathrm{T}$, $s,\overline{s} \in \mathrm{S}$ and $i \in \{R,C\}$, $U_i(s,t_1) \succeq_i U_i(\overline{s},t_2)$ iff $(s_i + t_1 c_i) \succeq_i (\overline{s}_i + t_2 c_i)$.

Assumption 5 (A5) – Stationarity: For $s,\overline{s} \in \mathrm{S}$ and $i \in \{R,C\}$, $(s,t) \succeq_i (\overline{s},t+1)$ iff $(s,0) \succeq_i (\overline{s},1)$.

3.5.5 Negotiation Strategy

In real world, the agents will not know how the opponent will behave. Moreover, each agent may have dynamic attitude towards risk during the negotiation process (i.e. due to changes of the AMS's policy, this value may be altered while the negotiation is in progress). In this case, we approach the solution algorithmically. We develop the agent's negotiation strategy which mimics natural human behaviour where the negotiator gets increasingly impatient as time passes since the negotiation cost rises while the probability of not gaining (or even losing) from this negotiation increases. We define an agent's patience as follows:

Definition 5 – Agent's patience: Agent i will concede more when the probability of not gaining from this negotiation increases. Let n_i be the smallest integer, for which the following inequality still holds:

$$n_i \le \frac{M - U_i^{\min}}{\sigma\delta + c_i} \tag{3.2}$$

where $U_i^{\min}$ is the minimum utility that i deems to be necessary to continue the negotiation. n_i reflects the patience of the agent whereby the lower it is, the more impatience the agent is.

From A1, when the agent gets increasingly impatient, it is more willing to concede more so as to entice his opponent into accepting its offer. Intuitively, the agent who is more impatient will be more inclined to concede and thus agreeing to an offer that is less advantageous to itself. Further, this agent is more likely to opt out of the negotiation since it approaches its minimum utility, $U_i^{\min}$, faster than its counterpart.

Definition 6 – Agent's concession function: The concession that i makes at time t is defined as:

$$\triangle_i(t) = f(t \mid n_i) = e^{\gamma_i\left(t/n_i - 1\right)} \tag{3.3}$$

where γ_i depicts i's risk aversion on the outcome. By this definition, the agent's concession increases exponentially over t for constant γ_i.

The parameter γ_i can be configured via policy by the specific AMS. From definition 5, i is said to be risk-averse when its γ_i is of a high positive value and conversely, it is considered risk-loving when it has a small γ_i value. Note that, since the concession is an exponential function, by Arrow-Pratt measure of absolute risk-aversion [40], the

AMS's attitude to risk is constant throughout the negotiation process as long as its value is not altered.

Definition 7 – Agent's negotiation strategies: The agent's negotiation strategy specifies the agent's next move chosen from the set $\mathrm{S} \cup \{ACCEPT, REJECT, OUT\}$.

We begin by defining the strategy for the agent who has to make an offer in the first move (i.e., R). It has to make an offer from set S when t is even and respond from the set $\{ACCEPT, REJECT, OUT\}$ when t is odd. When t is even, R makes an offer (α_R^t, β_R^t) based on the following:

$$\alpha_R^t = 1 - \Delta_R(t) \tag{3.4}$$

$$\beta_R^t = \Delta_R(t) \tag{3.5}$$

However, if $U_R(t) < U_R^{\min}$, the following relations are used:

$$\alpha_R^t = \frac{U_R^{\min} + tc_R}{\sigma\delta(dl - t)} \tag{3.6}$$

$$\beta_R^t = 1 - \alpha_R^t \tag{3.7}$$

When t is odd, R shall make a decision based on the latest offer proposed by C. If the utility computed from C's offer is greater than or equal to R's utility if R were to offer at this period (we refer this as the *expected utility*), then R will accept this and inform the OP about the decision. Otherwise, R will reject the offer and the negotiation proceeds to the next round. However, there is an overriding clause to this in which R will opt out if $1 - \Delta_R(t) \leq 0$.

Similarly, we define the strategy for the agent who has to respond to an offer in the first move (i.e. C in our case). This agent has to make an offer from set S when t is odd and respond from the set $\{ACCEPT, REJECT, OUT\}$ when t is even. When t is odd, C shall make an offer (α_C^t, β_C^t) based on the following:

$$\beta_C^t = 1 - \Delta_C(t) \tag{3.8}$$

$$\alpha_C^t = \Delta_C(t) \tag{3.9}$$

However, if $U_C(t) < U_C^{\min}$, the following equations are used:

$$\beta_C^t = \frac{U_C^{\min} + tc_C}{\sigma\delta(dl - t)} \tag{3.10}$$

$$\alpha_C^t = 1 - \beta_C^t \tag{3.11}$$

When t is even, C makes the decision based on the same logic as R, opting out only when $1 - \Delta_C(t) \leq 0$.

3.5.6 Case Study

This section illustrates the proposed negotiation framework through a case study. We consider a scenario where an OP governs a set of AMSs. Within this set of AMSs, R with $\gamma_R = 1$ has already established agreements with A_z, $z \in \{1, 2, \ldots, 5\}$ to route a specific amount of traffic separately through them to a destination AMS. Due to an unforeseen incident (e.g. a significant fault resulting in route failures), A_2 informs R that is forced to terminate its promised bandwidth to R but expects to return to normal service within a day. Consequently, R requires new bandwidth immediately to support its customers. We assume that failure to meet the service level agreements (SLAs) of the affected customers will result in a penalty of 3750 credits per minute to R. In order to preserve its reputation, R has to try and serve the committed SLAs the best it can during this interruption period. This implies that $U_R^{\min} = 0$.

Therefore, R sends a request, $REQ = \langle 2,\ BW,\ 150\text{Mbps},\ 1440\text{mins},\ 50\text{credit/Mbps/min} \rangle$ to the OP to initiate a negotiation. Essentially, R is requesting 150 Mbps of bandwidth for a duration of 24 hours. and the passing of each minute signifies the decrement of benefit of 50 credits from the customers.

Based on the latest Ad_z, OP lists those AMSs that have adequate resources to offer, eliminating those that do not. In this case study, the OP follows the greedy algorithm to choose the next AMS to negotiate. The qualified candidates are ranked in decreasing order as follows: A_3, A_1, A_5. A $START$ message is then sent to R and A_3.

Although A_3 is willing to negotiate the share of the requested bandwidth, it has an alternative use for them which would fetch some benefit. A_3 is taking some risks in attempting to barter its resources for higher gain. Thus, this negotiation involves a high opportunity cost for A_3. This is reflected by its properties: $\gamma_{A_3} = 10$, $U_{A_3}^{\min} = 9000000$ and $c_{A_3} = 40\text{credit/Mbps/min}$. We show in Figure 3-3 the evolution of the utilities for both agents in this negotiation. It should be noted that the expected and offered utilities never intersect. In fact, A_3 opts out at $t = 132$ when it is clear that this negotiation will not yield higher gain than choosing its alternative option.

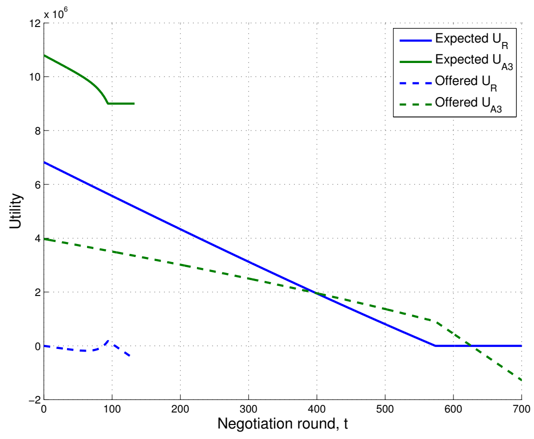


Figure 3-3. Negotiation between R and A_3 - expected and offered utilities.

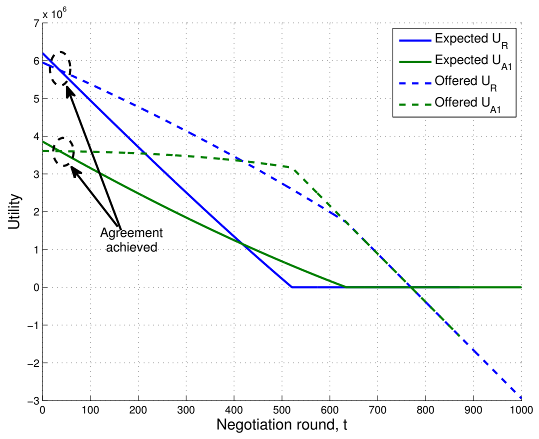


Figure 3-4. Negotiation between R and A_1 - expected and offered utilities.

The breakdown of this negotiation forces R to continue seeking for a new agreement. The next in line to negotiate is A_1 which is eager to strike this deal since otherwise, its spare resources will remain unutilized (i.e. $U_{A_1}^{\min} = 0$). Thus, A_1 chooses to avoid risk by setting $\gamma_{A_1} = 0.5$. Finally, the cost per negotiation round only accounts to the fixed maintenance cost i.e. $c_{A_1} = 10\text{credit/Mbps/min}$ as there is no opportunity cost for A_1. Figure 3-4 shows the utilities for both agents in this negotiation. At $t = 36$, A_1 accepts

R's offer of (0.6166, 0.3834) and achieves approximately a 60-40 split, with R getting the bigger share. Although in this negotiation, A_1 has the cost advantage over R, its conservative approach of not taking risk has worked to R's advantage.

3.6 Conclusions

This section presented the concept of the Orchestration Plane, which enables the cooperation of the various autonomic control loops in the network, ensuring operation within the boundaries set by the business goals defined by the operators.

Orchestration arises from the need to coordinate the operation of two or more autonomic control loops, which may reside at a single domain or at different domains. In the future Internet, each domain will be run by different autonomic management systems. Although management will be decentralized, a set of common inter-domain constraints and SLAs must be agreed upon in order to provide assurable QoS for end-to-end services.

The OP is realized by the Distributed Orchestration Components, which have two main components: The Dynamic Planner, which is a policy-based dispatcher of orchestration tasks, and the Behaviours, which model each specific orchestration task. This division allows for the extensibility of the architecture, since the planner is a generic component, taking care of the interaction of specialized components, the Behaviours. Behaviours are controlled by workflows, which are a set of steps and rules that define the interaction among behaviours, their triggering and shutting down conditions.

The latter part of this section presented a game theory bargaining approach for the bilateral negotiation between Autonomic Management Systems under the governance of an Orchestration Plane to achieve a specific collaboration. We illustrate how an OP may facilitate the negotiation process and foster collaboration of different self-interested entities in a highly dynamic setting of Future Internet Networks in a controlled manner. The main contribution of this work pertains to the negotiation process itself where we describe a negotiation framework based on the bilateral bargaining model of alternating offer. By first establishing the preference relationship between the agents and then using the backward induction method, we contribute to the solution of the new bargaining model under perfect information scenario. However, the AMSs often have no information about their counterparts in practice. Hence, we also contribute on solving the problem where both agents negotiate “in the dark”. We developed a bargaining algorithm where the agent makes an offer based on its patience and risk aversion on the outcome.

4 Policies for Autonomic Management (PUMA)

4.1 Introduction

The network's design is moving towards a different level of automation, autonomicity and self-management. The envisaged solution is based upon an optimized network and service layers solution which guarantees built-in orchestrated reliability, robustness, mobility, context, access, security, service support and self-management of the communication resources and services. It suggests a transition from a service agnostic Internet to a service-aware network, managing resources by applying autonomic principles.

A key element in the above conception is the elementary autonomic cell; this is, the minimum functionality to deploy exhibiting self-management properties. Our vision is that sets of these elementary cells would then be coordinated (orchestrated) to make a complete system or network self managed. This elementary cell is called Autonomic Management System (AMS) in the context of this work. We assume that the behaviour of an AMS is policy-based and the purpose is to investigate and provide a solution for the set of policies to drive the behaviour of AMSs.

We start this section providing a background on our conception of an AMS. This includes an architectural approach specially depicted in order to highlight the impact of policies in the system architecture. Then we continue presenting the means to represent policies and the types of policies are we interested about. Finally, we propose the specific sets of policies to cope with the main aspect of the AMS lifecycle.

4.2 The Autonomic Management System Concept[3]

The AMS is an autonomous infrastructure that manages a particular network domain, which may be an Autonomous System (AS) (i.e., the Autonomous Systems identified in RFC 1930). AMS implements its own control loop(s), consisting of context collection, analysis, decision making and decision enforcement. Mapping logic enables the data stored in models to be transformed into knowledge and combined with knowledge stored in ontologies to provide a context-sensitive assessment of the operation of one or more virtual resources. The AMSs communicate through sets of interfaces that: (i) enable management service deployment and configuration (i.e., the ANPI & vSPI interfaces, defined in [27]), (ii) manipulate virtual resources (i.e., the vCPI interface, defined in [41]).

Since the AMSs implement their own control loops they can have their own goals. However, their goals should be harmonized to the high-level goals coming from the DOC that is responsible for each particular AMS. Each DOC is responsible for a set of AMSs that form a network domain, called Orchestrated Domain (OD). An OD may belong to a single organization that has the same high-level goals. We note that the entry point for the high-level goals is the Orchestration Plane. For example, AMSs may re-establish a local service in case of failure without interacting with the Orchestration Plane. However, this new establishment should follow the same guidelines that this local service used to follow. These guidelines are defined from the Orchestration Plane. So, there is a significant difference between management and orchestration.

[3] The background has been adopted from the Autonomic Internet project

Orchestration, actually, harmonizes the different management components to one or more common goals.

An AMS is realized through a set of virtual routers and management software, implementing one or more self-* capabilities. The software running on the virtual routers constitutes the autonomic control loop. In Figure 4-1, we give an example of an AMS. The virtual routers depicted are actually the virtual resources that are managed from the particular AMS. These entities have embedded management functionality in the form of management components. For example, components like a knowledge plane component (e.g., a CISP node), a policy decision point etc. We note that this example is simplified in the sense that does depict all required components for a complete control loop, which are required for the establishment of an AMS.

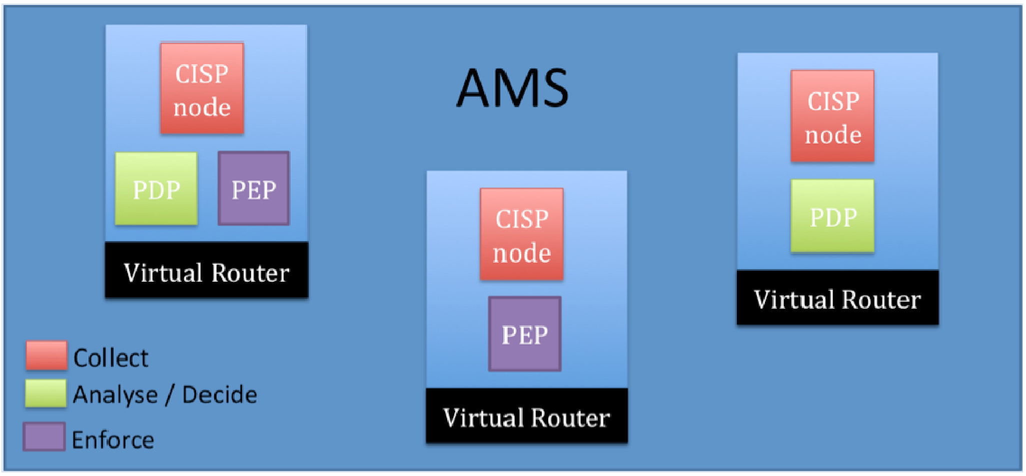


Figure 4-1. The Autonomic Management System (AMS).

We assume that a virtual router is the network entity of the Future Internet, having embedded network management capabilities. We need to draw a line between the management functionality that needs to be embedded to the devices and the management functionality that does not. In this context, a virtual router may be demonstrated by a XEN Virtual Machine.

4.3 Policy-based AMSs with Reasoning Capabilities

Each AMS implements a number of control loops in order to exhibit self-* capabilities. At this stage of the project, we assume that the control loops are policy-based; therefore, this section describes the components of a policy-based AMS.

A policy-based AMS is similar to a traditional policy-based management system except that in addition to the standard functionalities, it should be designed to allow for each of its components (including its policies) to be influenced by a DOC, and in that it should be able to operate with entities outside the scope of its administrative domain. The AMSs should also influence their policies changing them based on learning. The key components of the policy-based AMS are described hereafter, in relation with the components of the management model defined in [25].

The autonomic control loop in the AMS is realized by using a control loop inspired by the FOCALE architecture [42]. The control loop as depicted in Figure 4-2 requires sensors or monitoring applications that can determine the current state and respective context of the managed resources. This sensed information is translated (via model based translation) and analyzed. The analysis of the state and context information is carried out for each managed resource and is an important step in the control loop. Here, the analysis process investigates if the state of the managed resources is currently in a “desired” state. If so then no action is appropriate and the control loop passes control back to the monitoring applications. However, if the analysis process determines that the managed resources are in an “undesired” state then an appropriate action should be taken with the objective of causing the managed resources to reach a desired state. The appropriate actions are enforced onto the managed resources using the concepts of actuators and effectors.

Each step in the control loop, namely sense, analyze and act, is policy-enabled. Therefore, the type of data being gathered to represent the state and context of managed entities is changeable. Also the algorithms being used to determine desired state and what constitutes a desired state can be changed. Finally, the method of devising a strategy to find and enter a desired state can also be changed. We note that these dynamic components need to be gradually deployed and tested. The stability of the proposed system is a main goal of this work. To ensure that the most appropriate policies are used to define the behaviour of each step in the process, reasoning and learning processes are incorporated to take into consideration historical data, experiences and inferred information. Policies are changed according to this learning, when it is necessary. To support this control loop, an enhanced policy-based system is used to realize the AMS.

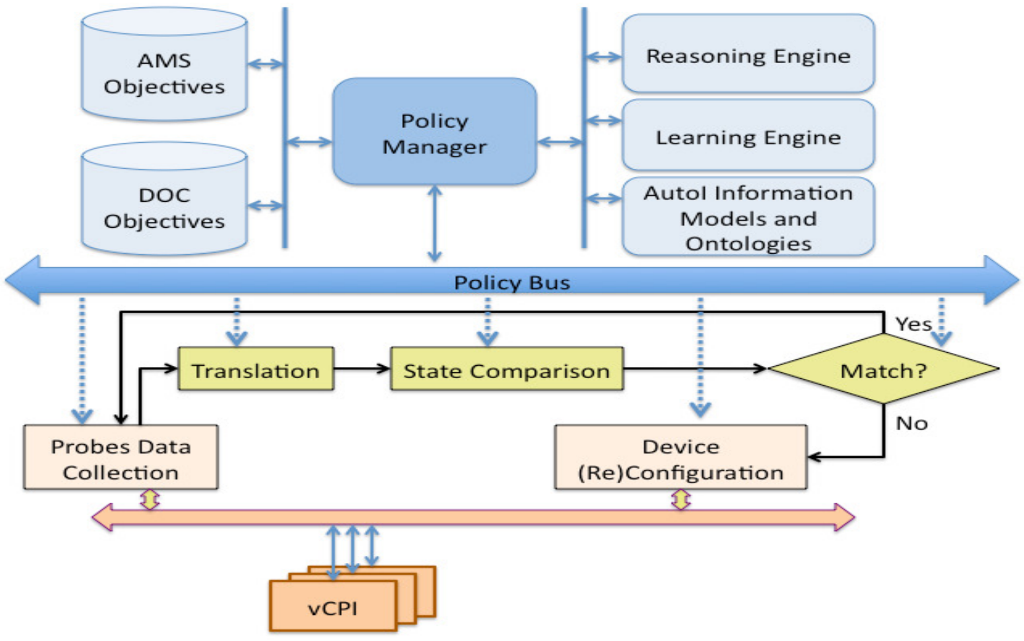


Figure 4-2. AMS control loop.

Representing Policies in the AMS

As each AMS may be developed to manage very different applications, services and resources; the policies used may have different definitions and characteristics. Therefore, the policy model in use is designed for maximum flexibility, while keeping expressivity in mind. ECA (Event-Condition-Action) policies can be defined, along with Access Control, Utility Function and Goal policies. There will be a specific language for representing the structure of policies for the various applications. These policies can then be analyzed using the analysis process, and once they are deemed valid and conflict free, they can be deployed directly or transformed into third party policy representations.

4.4 Policy Continuum usage

Policies deployed as a policy continuum are separated into multiple levels of concern. This is typically represented as concerns related to Business, System, Network, Device and Instance aspects of an organization. Business policies are at the highest level and Instance policies are at the lowest level. Business policies may also be referred to as High-level policies.

The policy continuum is described informally in [43], where there is no description of how policies should be represented at the various levels, or how the modification of those policies should be managed. This has been addressed in [44] where a formal model of the policy continuum is proposed, along with an authoring process that formally describes how an individual policy can be added or removed from the policy continuum. This authoring process comprises of three main phases: policy verification, policy conflict analysis and policy refinement.

To date there has been a lot of research in the all three areas, but there exists no work that attempts to address all three with respect to an authoring process for a policy continuum for autonomic communications.

The challenge to describe the Policy Continuum entails:

- Deciding on a representation of policies.
- Deciding on the number of levels to the Policy Continuum.
- Identifying some initial methods of policy verification, policy conflict analysis and policy refinement.
- Outlining the authoring process for policies in the Policy Continuum.

In the following we outline the first two challenges.

4.4.1 Representing Policies

The representation of policies will follow the information model as presented in [25]. This model is flexible to suit the needs of policies at any level of detail required. Once defined, the policies can be readily deployed into policy applications, or transformed into third party policy languages for deployment. The types of policies that can be represented are Goal, UtilityFunction and ECA policies.

4.4.2 Number of APC Levels

Considering that the system architecture is not focused on a single administrative domain, there is a need to slightly change the concepts of the levels of policies.

4.4.2.1 High Level Policies

These policies represent the business objectives and goals of a single AMS, or administrative domain. Therefore, there may be many high level policies that will remain local to a single AMS, or shared across multiple AMSs. High-level policies can be related to services that the organization offers, or is willing to host for other domains. AMSs can be federated. This can be manifested as two of more AMSs agreeing (via the DOC) to abide by a set of common business objectives or high-level policies. The federated AMSs may be established to serve the end-to-end deployment of a service or to improve the reliability of communication links for the participating members. Goal policies are the most appropriate form of policies for this use.

High level policies could cover for instance the way by means of which the resources available at a given node (i.e. memory, CPU, input/output connectivity) will be split and sold to different virtual operators.

4.4.2.2 Orchestration Level Policies

These policies are related to high-level policies, as the decisions outlined by high-level policies impact the decisions being made by orchestration policies. These policies control the behaviour of negotiations between domains, the distribution of tasks and information between domains and the agreements for federation between domains. These policies are used to inform the DOCs about decisions it can take on behalf of AMSs with respect to federation, negotiation and distribution. Deontic and ECA policies are the most appropriate form of polices for use here. Deontic policies are used specifically when multiple systems need to coordinate, for example, in exchanging information (authorization, prohibition) or for giving orders (obligation, dispensation). ECA policies are reaction based polices so that a DOC can react to the decisions made by the AMSs.

4.4.2.3 System Level Policies

These policies are related to the management of an AMS, which distributes management tasks over virtual routers and services. An AMS has a two-tier management architecture, where the AMS can be managed using policies, and individual virtual routers in the AMS can be managed using policies. These policies populate the policy-based AMSs as described in the previous section. These policies are associated to high-level policies in that the decisions that can be made at this level are directly impacted by the decision made at the high level. For example, a high-level policy may introduce precedence into the use of technologies to support encryption depending on the type of service being delivered. Given this precedence, the system policies will be compelled to follow it as best it can.

These policies are used to aid in realizing the autonomic control loop and therefore are used to manage and occupy the steps of the control loop.

System level policies can decide for instance how the available resources assigned to a given virtual operator (VO) are used to support the services of this VO. An example of that is a policy that decides to force the sharing of the same resources (virtual routers) to connectivity services of the same protocol (i.e. IP v4) whereas it forces the creation of different virtual resources when a service supported by a different protocol is requested. Also, a good example of this group of policies is to decide the behaviour of the AMS in case of QoS degradation of supported services.

4.4.2.4 Component Level Policies

These are policies defined to manage individual Components as defined in the AutoI architecture [26]. The policies can be associated to managing virtual resources and management services deployed onto the component. These policies are potentially optional if it is sufficient to manage these resources from the system level.

Policies specifically devoted to manage intra-domain access points could be a representative example of this group of policies.

4.4.2.5 Instance Level Policies

These policies are instance level policies that are embedded inside devices that can perform their own decision functions. An example of this is routing functionality where the decisions need to be performed on a per packet basis. Policies defined at a higher level may make decision to re-configure devices capable of hosting instance level policies.

The concept of instance is perhaps better realized in the context of physical resources. For example we can talk about the component ”router” and a Cisco 7000 would be a specific instance of that router component. Also, talking about platforms supporting virtual routers we could distinguish between the platform as a component and a given physical machine as the instance of that component.

4.5 Proposal of Specific Policies for AMS Behaviour Control

4.5.1 High Level Policies

As soon as an AMS is booted it must distribute its available resources among N potential customers (service providers). This is done by instantiating one virtual router per service provider and assigning a given amount of memory, CPU usage and bandwidth to it. Figure 4-3 shows two physical nodes belonging to a network provider which is wishing to offer its resources via virtualization. Each node is characterized by a given amount of memory and different CPUs (two in each node in the Figure 3). Also note that each node contains communication cards with a given amount of bandwidth. On the left hand side, these cards will be used to grant access to users of the different service providers, whereas the other cards are to communicate the two nodes and these ones to other domains.

Figure 4-4 shows an example of what can be the consequences of the high level policies. After the AMS booting process the resources in both nodes are split in order to eventually create three virtual networks. The specific amount of resources assigned to each service provider / virtual network is according to these high level policies as well. In the case of the example we observe that one CPU core has been assigned to one virtual network whereas the other core is shared between the other two networks.

Finally, Figure 4-5 is like Figure 4-4 but from the connectivity point of view. Looking at virtual network 1 (VN1) for instance we can say that it is constituted by one virtual router VR_{a1} in node A that has to accomplish its routing tasks with a maximum assigned memory M_{a1} and making use of CPU core C_{a1}. This router has to establish internal virtual links between ingress/egress access points characterized by a maximum bandwidth of B_{a11} and B_{a21}. Something similar happens in node B except that in this case there are two ingress/egress access points to connect to domains A and B with capacities B_{b21A} and B_{b21B} respectively.

We assume that each service provider has to commit specific SLAs with their customers. These SLAs would specify, among others, the transmission rate and the QoS. The QoS would likely be one of the standard DiffServ QoS. Nevertheless, for simplicity we will assume bit rate and packet losses to characterize the quality of service.

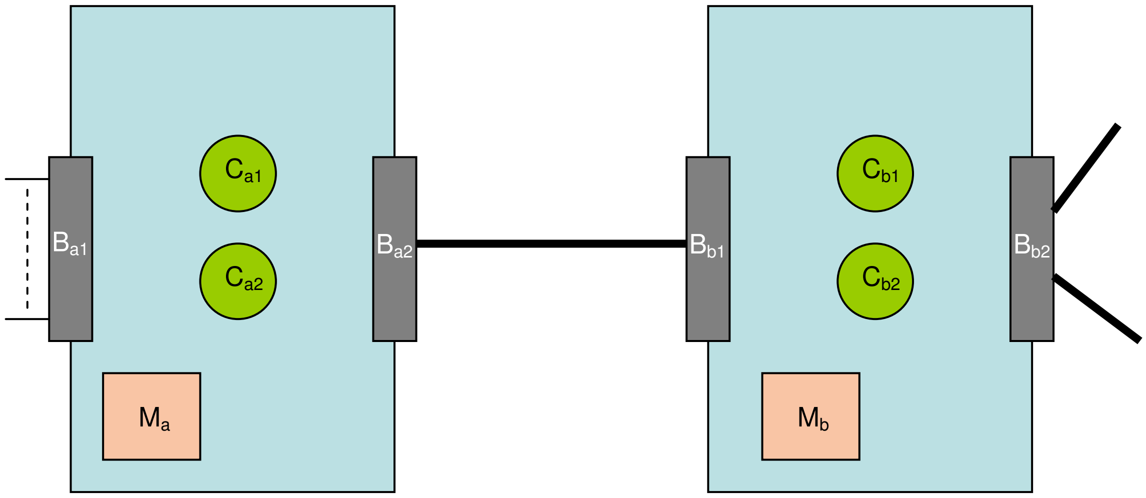


Figure 4-3. Physical resources in a given management domain.

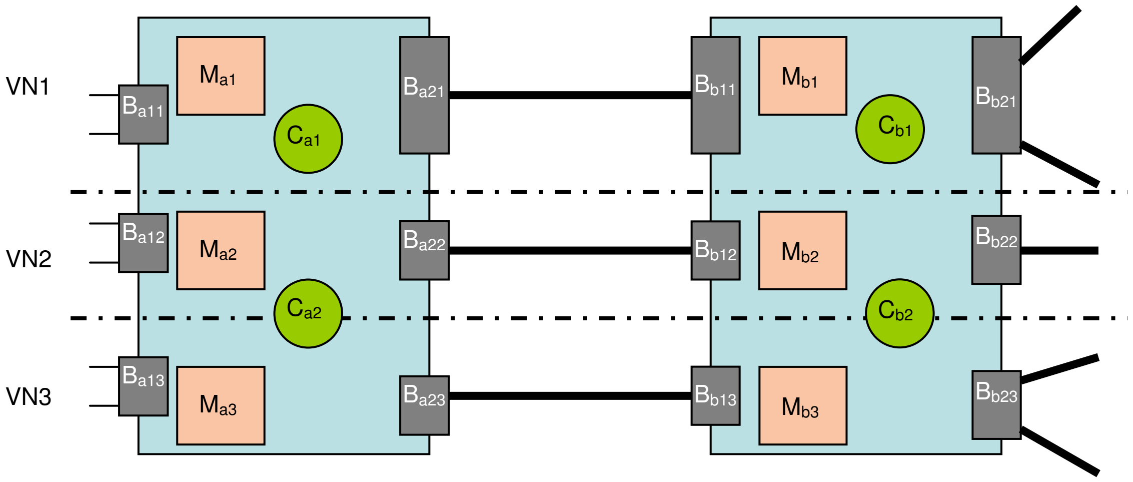


Figure 4-4. Physical resources after virtualization supporting three virtual networks.

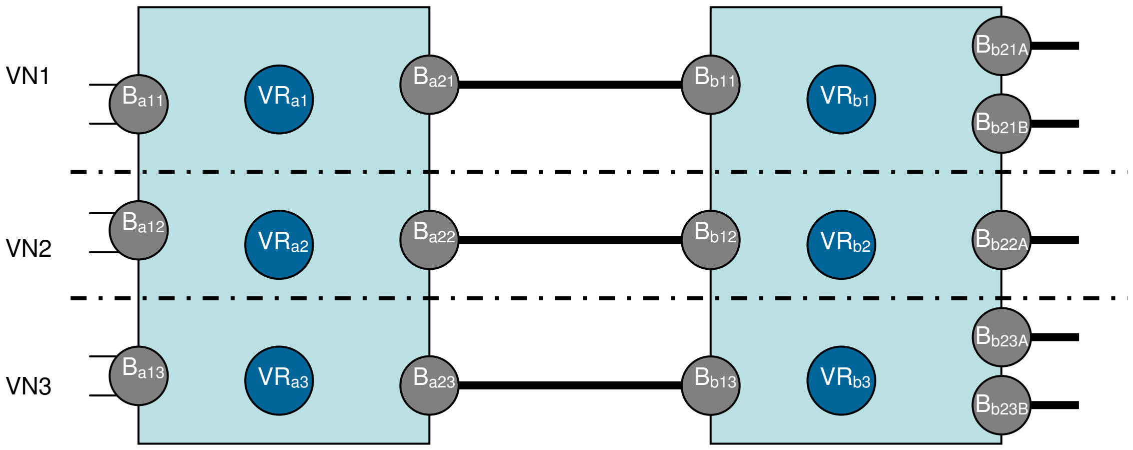


Figure 4-5. Virtual networks under the connectivity services point of view.

4.5.2 System Level Policies

System level policies will deal with the establishment of connectivity services as well as to tackle the situation of QoS degradation.

Once the AMS receives a service request, it offers that service with the currently available resources. For example, assume that an IPv4 64 kb/s service is requested to connect a user of VN1 to neighbor domain A. Then the virtual link represented in Figure 4-6 is established and the capacity of the resources involved is decreased accordingly.

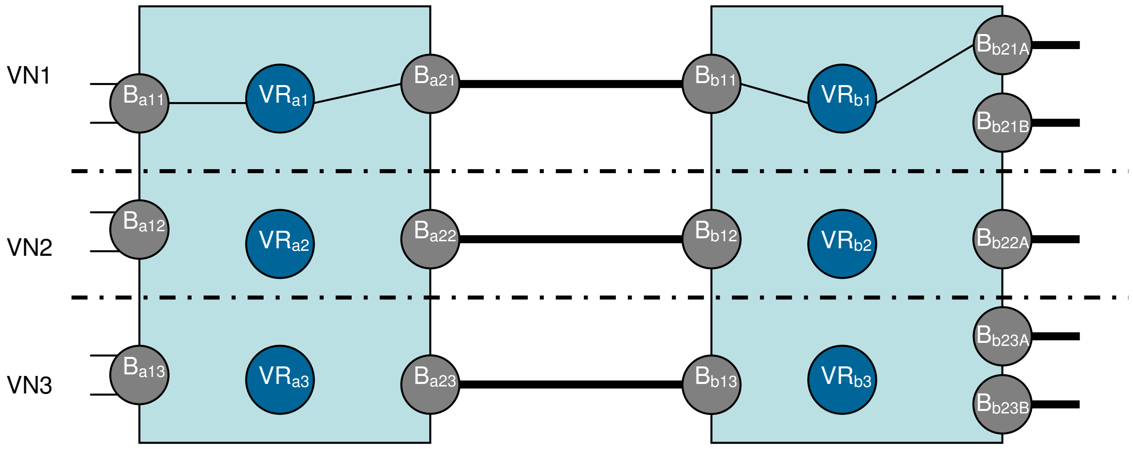


Figure 4-6. VN1 supporting connectivity services to neighbor domain A.

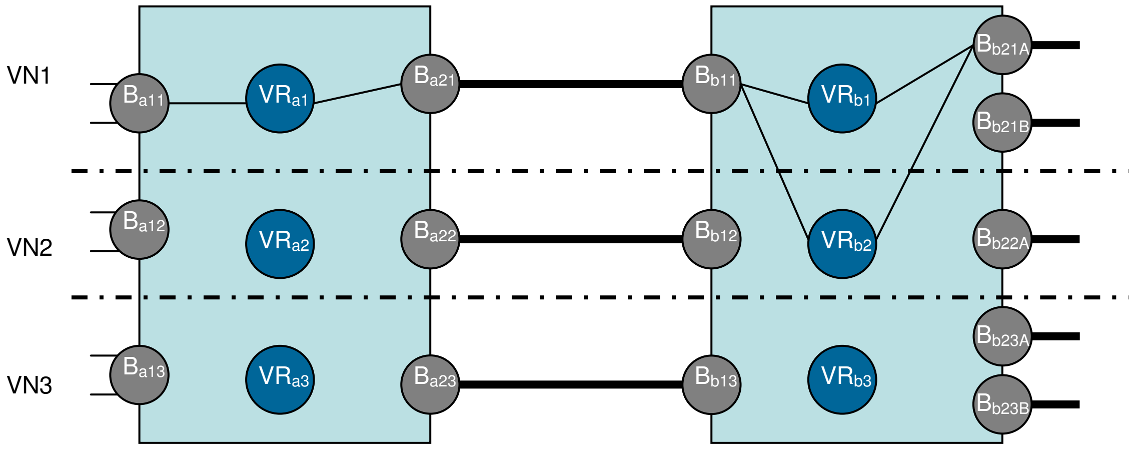


Figure 4-7. Service migration from VN1 to VN2 due to congestion in the former.

All other connectivity services coming after, will share the same access points and virtual links. The AMS will accept new services until the capacity of resources is near exhaustion or the QoS of the virtual network falls below a given threshold.

QoS degradation can occur well before the maximum link capacity is reached. One potential cause is the change in the distribution of packet's length. In fact, for the same transmission rate, shorter packets require more CPU processing time than longer ones and this can cause CPU exhaustion. QoS will be tracked through monitoring the virtual routers of the network. As soon as packet losses above a given value are detected in any router of the end-to-end link, system level policies will trigger a recovery process. This process may entail a reassignment of resources increasing the affected router or a migration of part of the supported services to spare resources. This means that the high level policies above mentioned have to take into account to reserve some spare capacity for eventual QoS degradation recovery. For that reason we assume that VN2 is a pool of spare resources instead of a network to be sold to a given service provider. Figure 4-7 shows a service migration from VR_{b1} to VR_{b2} due to congestion of the former.

4.5.3 Testing approach

Policies above described have to be tested in a real testbed scenario. We assume that this testbed is also a virtual laboratory of which we don't know precisely the amount of available physical resources. We also assume that we are allowed to instantiate as many virtual routers as needed. In that case we will precede assigning arbitrary values to the different variables involved by our policies and we will also emulate the congestion events. Accordingly, we propose to use access points of 1024 Kb/s and packet lengths of 1024 bytes. In addition we will assign 1 Mbyte of memory to each virtual router.

4.6 Conclusions

Policies for an AMS can be described by means of a Policy Continuum; this is a set of different interrelated policy layers covering different levels of abstraction. In principle we identified a policy continuum with five layers of granularity, namely high level, orchestration level, system level, component level and instance level. Out of these five we have identified the high level and the system level policies as the core set and therefore we have proposed specific policies to support both layers. High level policies establish how the total system resources (i.e. a computation platform) are split in virtual subnetworks to be sold to service providers. System level policies establish how each virtual network reacts under to request of a service and how it behaves under a potential degradation of QoS.

Work now is centered in the understanding of the potentials of ontologies to allow AMS with reasoning capabilities. These intelligent entities will constitute our second generation of AMSs.

5 Autonomic Management of Limited Capability Devices (AMICAD)

5.1 Introduction

The development of small wireless sensors and smart-phones which include various sound, video, motion and location sensors have facilitated new pervasive applications. These include healthcare, which requires monitoring body readings as well as activity; traffic condition estimation using GPS and accelerometers; monitoring and controlling temperature, humidity and lighting levels in buildings; environmental monitoring and flood warning and even tracking wildlife movement. These pervasive systems are expected to perform in a vast number of environments, ranging from urban to rural, with different requirements and resources. They are often mobile and the environment and application requirements may change dynamically requiring flexible adaptation. Users may be non-technical so the systems need to be self-managing. Some applications such as healthcare and flood warning may be life-critical so self-healing with respect to faults and errors is important.

Wireless Sensor Networks (WSNs) are closely related to mobile ad-hoc networks due to sharing the same general problems. Nonetheless, there are some differences between the two concepts, mainly with respect to energy considerations. In contrast to mobile ad-hoc networks, WSN nodes have fewer resources such as non-rechargeable batteries, less memory, slower CPUs and so on. The lifetime of a WSN can be shortened rapidly by misbehaving nodes or attacks on the network. Security aspects such as self-protection help to increase the operational readiness and secure the network against attacks. Former discussed concepts focused on self-configuration due to the complexity of large scale networks. To follow up on these, AMICAD investigates self-management concepts for WSNs in the context of security.

Another dimension of this activity is the use of policies as the enabling technology for automating the management of devices with limited capabilities. The Ponder2 policy framework, which has been specifically designed for small devices, has been used for the management of both WSN nodes and Internet tablets.

5.2 Self-healing for Wireless Sensor Networks

Typical pervasive computing devices have limited processing and power resources and a few sensors specific to the application. They have to cater for deterioration of sensor accuracy over time due to physical phenomena such as overheating or chemical fouling as well as external factors such as low battery levels or physical damage. Quality of wireless links may vary particularly with mobile systems and devices may completely fail. Faults in pervasive systems are considered more frequent compared to traditional distributed systems so provisions for handling and recovering from faults must be an important design consideration.

Frequent replacement of devices and manual recalibration are impractical and hinder the adoption and use of such pervasive systems by non-expert users with limited technical skills. The system has to be self-diagnosing and self-healing to maintain itself in a functioning state and mask component failures from users.

5.2.1 Architecture of a Wireless Sensor Network

Sensor networks can typically be structured in three distinct layers as illustrated in Figure 5-1 that presents the functional architecture of a wireless sensor network (WSN) application. At the bottom is the sensing layer, e.g. room temperature sensing for air-conditioning control in a building. The middle analysis layer is where sensing events are processed for making decisions of current context based on observations. For instance, readings from different thermometers can be collected at an aggregation point, typically a cluster leader, which combines multiple readings to determine a mean value, rate of change etc. Furthermore, decisions in local areas can be used in higher level. This is the decision fusion process where localized or low level decisions are combined along with application-specific knowledge for obtaining higher level perspectives.

The highest layer is the dissemination of decisions from the analysis centres to the network. This, typically, involve orchestration of information available in different parts of the network. Faults in this architecture propagate upwards affecting the decision quality in higher layers. As a result failures have to be handled as close to their source as possible.

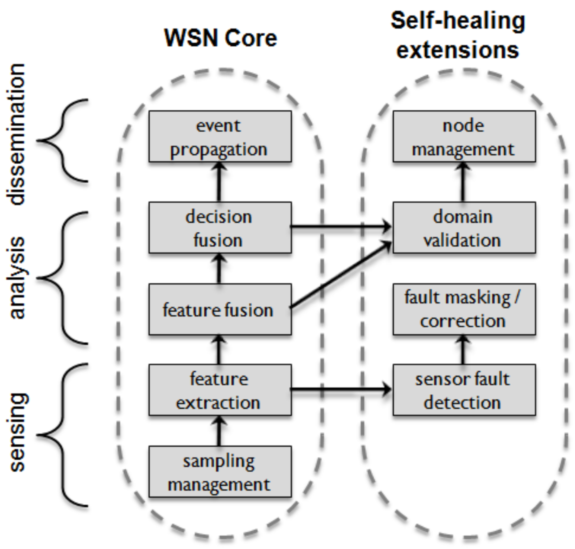


Figure 5-1. Sensing layers in WSN applications.

Pervasive systems should incorporate autonomic computing concepts in order to become self-manageable and minimize required manual administration by end-users. The four key attributes of autonomic systems are – self-configuration, self-healing, self-optimization and self-protection. These attributes are not orthogonal and some depend on others. For instance, in order for a network to heal a defective component should be able to reconfigure itself and possibly optimize the use of its resources to allow for backup solutions.

5.2.2 Fault Classification

We give a simple taxonomy of the faults we consider on the sensing devices. We define four classes of faults namely – short, noise, const and accumulative faults. Short faults are a momentary irregularity in the readings of a sensor, i.e. random spikes in the trace. Noise is a prolonged increased variance in the readings of a sensor. Const fault is an invariant repetition of an arbitrary value that is uncorrelated to the observed phenomenon. Finally, accumulative fault is a typically smooth, consistent deviation of the observed value from the ground truth, i.e. the sensor exhibits drift.

In order to evaluate the impact of faults in the original experiments we injected faults in the trace for each of the fault classes identified in Section 5.2. There are pros and cons associated with this approach. We have to create carefully the fault models to resemble real world faults, but artificially injected faults give a definite distinction between erroneous and normal behaviour, enabling evaluation of the detection accuracy of our mechanisms. As a result, we are more confident on the results of the fault detection techniques. A description of our fault models follows.

Short faults are inserted in the trace at a random percentage, *p*, of data-points by multiplying the existing reading by [+/-]*c* which is the fault intensity parameter.

For adding *noise* in a region of the trace, we use a Gaussian distribution with a mean of the original feature values and set the standard deviation as parameter *σ*.

Const fault is simply modeled by setting the value of a region to a specific value *v*.

Accumulative error is injected by multiplying the original value by *f(t)*, where *t* is the time and *f* is a function that monotonically increases or decreases with time and indicates the rate of drift.

We study the impact of each fault class by injecting faults in activity recognition (eAR) and gesture recognition (glove) traces. For every body sensor node (BSN) we inject fault into 1/3 of the signals. Accuracy decline in overall classification is illustrated in Figure 5-2. For the glove experiment, we present impact both on each node individually and when combining all inputs from the nodes. Where classification should be implemented is a tradeoff between data transmission and computation. In essence, whether classification takes place on the nodes without any raw data transmission or in the base station, which collects all the readings from the nodes.

For short faults, the impact on the accuracy is minimal, especially when we consider all 20 inputs in the glove experiment, where there is almost no degradation. For the short class we have set the occurrence of faults to 5% while the intensity of the fault varies from 10-40%.

Noise has greater impact on the individual nodes, but again in the case where we consider all inputs of the glove trace the decline is small. Ample redundancy of inputs seems to compensate for inaccurate data, even though input measures different attributes. It is also worth noting that the eAR node and BSN3 are affected in a very similar manner though they are used in completely different setups. This is apparently attributed to the fact that they are both equipped with only three sensors and are more sensitive to the loss of 1/3 of their input compared to the rest nodes.

Accumulative faults on the other hand present a more severe impact on the classification process. Even when considering all the glove inputs we drop to 70% successful classification from the original 90%. Another interesting finding is that the eAR trace has significantly lower tolerance to this class of faults, compared to individual nodes of the glove trace. This illustrates that the impact of faults depends on the monitored phenomenon, in addition to the type of sensor, in which case are identical for both case studies.

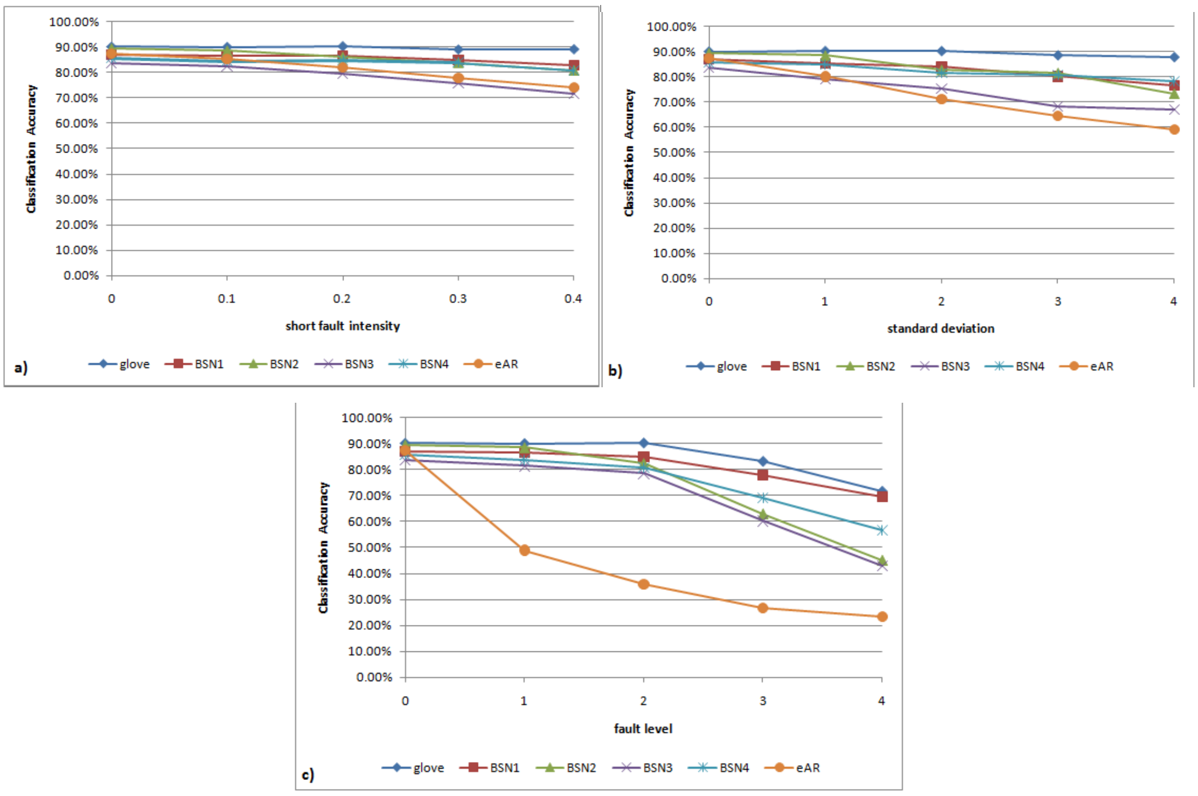


Figure 5-2. Classification accuracy degradation for a) short, b) noise and c) accumulative faults

5.2.3 Fault Detection

Results demonstrate how the classification accuracy is affected by faulty signals in the trace. In most cases we gain by removing a faulty signal instead of letting it confuse the classification process. Consequently, we investigated fault detection methods to identify faults during the system operation and provide the input for triggering reconfiguration of the network.

Short faults manifest themselves as an irrelevant sharp increase or decrease of the sensor’s value compared to its given history. If we consider the sensor’s recent input history as a Gaussian distribution, the values within three times the standard deviation distance from the mean value account for the 99.7% of the points. We regard a short fault as a data-point whose value exceeds that limit.

Const faults are trivially identified as multiple readings with the same value i.e. a variance of zero, which would be unusual behaviour for an accelerometer. Other sensors with a more static behaviour of reading e.g. temperature would require methods of neighborhood correlation similar to those employed for noise and accumulative faults.

For noise error, increased variance in the trace does not work as activity changes inherently result in large variance, although this may be suitable for other types of sensors such as thermometers. We use a voting scheme between the sensors to distinguish between a faulty region and activity change variance. If a minority of the associated signals indicates large variance, this is likely to be due to faults.

Even though sensors monitor different attributes of a phenomenon, these attributes are interlinked allowing us to exploit this information for checking the state of the sensing devices. However, attempting to set a static threshold does not generalize as a pattern for detection in different scenarios. Attempts to apply an online learning algorithm for an adaptive threshold did not provide fruitful results, yielding many false positives.

In the case of accumulative faults, i.e. drift, the variance feature does not provide enough information for identification, in contrast to the other classes of faults. Instead, we study the trend line that the readings' history forms using regression methods. By calculating the trend of the data, we can exploit domain knowledge to evaluate their validity. In the accelerometer case, the trend of readings resembles a line parallel to the x-axis, while sharp changes appear in all axes during activity changes. A voting scheme is again being used to distinguish between normal behaviour and faulty regions.

The methods described above for noise and accumulative faults require specific knowledge of the domain in order to set effective threshold of high variance or expected line slopes. Experimenting with online learning and extracting threshold from neighbors, indicated that they did not apply in our scenarios, each for a different reason. A self-adapting threshold from history of readings does not work with accelerometer readings with activity changes, which inherently changes the sensors' output and hence modifies features. In the scenarios where state changes are frequent, online knowledge is outdated very quickly and misleads the system. A threshold that is extracted at a given time is usually only be appropriate to the current activity and as soon as the activity changes, it yields false alarms of sensor misbehaviour.

Instead, we use the correlation feature between two random variables, in our case the pairs of sensor signals, as a metric for fault detection. The correlation between two random variables *X* and *Y* is defined as:

$$corr(X,Y) = \frac{E(XY) - E(X)E(Y)}{\sqrt{E(X^2) - E^2(X)}\sqrt{E(Y^2) - E^2(Y)}} \quad (5.1)$$

E(X) in (5.1) is the mean value of the random variable X. Correlation is bound between the values -1 to 1. The closer the absolute value of correlation is to 1 the more correlated the random variables are, where negative sign refers to negative correlation. A value of 0 denotes that the variables are completely independent.

We expect the signals in a BSN to have a high value of correlation. Therefore, we assume that low levels of correlation are not a normal behaviour in the system and trigger an alert for faults in the sensors. The fault that is detected can fall in any of the const, noise or accumulative classes.

5.2.4 Experimental Results

The metrics we use are the *hit-rate* of detected faults, where a hit indicates at least one alert is triggered when a fault occurs; the *fall-out* which is defined as the ratio of false positives to the sum of false positives and true negatives; and the time *delay* between a fault occurrence and its detection measured in samples, where on average there are 32 samples per second in the traces. The duration of an injected fault is approximately 1200 samples. Table 5-1 summarizes our findings for detection of short, noise and accumulative faults. We omit evaluation for the const fault technique. As we have discussed earlier, their detection in our case studies is trivial.

The model of the short fault proves to be very accurate, yielding very low false positives. The delay attribute is not relevant for short faults as they are instantaneous. In Table 5-1 we compare the variance (v) and regression (r) with the correlation (c) techniques for noise and accumulative faults respectively. Both noise detection techniques are very accurate on detecting actual, noise. However, the variance method gives slightly increased number false positives, compared to the correlation technique. On the other hand, noisy areas are detected quicker. The increased delay is attributed to the larger history of samples used for the correlation method. Consequently, the feature adapts slower to new behaviours. The history size is a tradeoff between detection delay and the number of false positives. We used a history of 200 samples in the experiments.

Table 5-1. Detection Method Evaluation

	hit-rate	fall-out	Delay
short	99.54%	0.77%	0.00
noise (v)	98.33%	7.43%	8.41
noise (c)	100%	6.32%	143.53
accum (r)	95.22%	19.47%	197.99
accum (c)	78.22%	3.89%	346.31

The regression technique for accumulative faults has a higher hit-rate comparing to correlation, but at the cost of a high fall-out. As expected, closer examination of the results indicated that the drift cases that escaped detection from correlation are those that have a very smooth deviation from ground truth. The higher detection delay of the accumulative errors is tolerable in the drift case, as the effects at the beginning of its appearance are marginal. Regression analysis for detecting the trend of the input is a very computationally intensive process. An alternative is a very rough estimation by calculating the slope of the line passing over two data-points. This approach yielded slightly degraded results compared to regression, but still comparable, considering the computation gain.

The greater benefit from using the correlation technique for detection is that it is a more generic method compared to variance and regression that require specification of custom-tailored thresholds per deployment. Furthermore, the correlation technique is more resilient when more than one class of faults is present.

5.2.5 Adaptation on the Nodes

It becomes evident that adaptation is required on the nodes as theirs and neighbouring sensing capabilities are modified over time. Policies have been studied and applied for management of traditional distributed systems to allow administrators cope with the complexity of distributed systems. Ponder2 is a policy system that has been designed with scalability in mind; scaling from mobile devices that support a Java runtime to large-scale distributed systems.

However, motes typically used for sensing devices are not able to execute the Java runtime. We developed a simplified version of Ponder2, with limited expressiveness, called Finger as a policy middleware for TinyOS motes for healthcare monitoring. The Finger2 system, described here, is an evolution of the original Finger system that has been ported to TinyOS 2.x, along with enhancements in the expressiveness of the policies.

```
def auth_policy[+/-]
      subject role
      target resource
      action something

def oblig_policy
      on event
      if condition
      do action
```

Figure 5-3. Template authorization and obligation policies

Ponder2 supports two kinds of policies. Authorization policies are access rules controlling the conditions under which subjects are permitted to (or prohibited from) accessing target services or resources. Obligations are event-condition-action rules, which can be used to specify adaptive management strategy. Templates of both kinds of policies are presented in Figure 5-3.

Finger2 obligation policies can control the behaviour of individual nodes allowing a constrained form of dynamic re-programmability. Policies are designed to be lightweight and compact, to transmit over the air, allowing motes to adapt to new conditions without requiring them to reboot. On the other hand, updating policies does not enable us to completely reprogram a mote or adding new functionality that is not already available in its ROM.

TinyOS is an event based operating system that fits well with the concept of obligation policies that are activated as responses to events. Events in the case of Finger2 are either internal originating from the node itself or external; coming from the network. Events also carry context, i.e. parameters, which can be referenced inside a policy, have their values checked or passed for an action evaluation. nesC is the native programming language of TinyOS. In essence, it is the C programming language with some component oriented extensions that promote code reuse and clean architectural structure of the embedded code. Furthermore, nesC constrains the use of some C features like function pointers, recursion and dynamic memory allocation, i.e. everything lives in the stack. A side effect of this is the ability to statically reason about program attributes such as maximum stack size, at compile-time.

The design is easily extensible simplifying integration to Finger2 for application developers providing three basic interfaces of which application developers should be aware. These interfaces are the main hooks for integration with the policy system, when a specialized component is required that is not already provided in the framework. In the following section we give an overview of the provided components in the system.

Finger2 provides a component library to support the most commonly used functions in a WSN application. This includes sensor sampling, buffering and feature extraction facilities, timers for scheduling of events and network primitives for exchange of messages among nodes. Additionally, the Serial component enables transmission of data over the serial port for application logging and debugging purposes. It, further, allows the application to accept messages and events from a desktop client.

The policy management is also exposed as a library component giving full control of policies on nodes, such as enabling/disabling or dynamically updating them. This is the key feature that allows adaptation in the network as nodes can modify their policies as a form of constrained reprogramming without disturbing node’s operation. The reprogramming is limited in policies, as it is no possible to load new actions or events, i.e. TinyOS does not support dynamic modification of native code on the motes. Native code loading would require in-place re-flashing a new image in the ROM.

Policies can interact with the aforementioned components by calling the actions or predicates and registering for event notification they provide. Functions or algorithms that are more specialized can be implemented in a similar fashion as modules that integrate with the policy system through simple interfaces.

5.3 Security Enhancement for Wireless Sensor Nodes

Wireless Sensor Network (WSN) nodes in contrast to MANET nodes have only limited resources like mostly non-rechargeable batteries, fewer memory, slower CPUs and so on. The lifetime of a WSN can be shortened rapidly by misbehaving nodes or attacks on the network. Security aspects like self-healing and self-protection help to increase the operational readiness and protect the network against attacks. Self-management concepts for WSN nodes in the context of security are in the focus of this work. Concepts taken from the well-known Autonomic Computing research field as well as virtualization concepts are so far not applied in WSNs. We believe that these concepts are the key for developing further secure and reliable sensor nodes, operating systems, and applications of the next generation.

Wireless sensor networks have some research overlap with other wireless networks like MANETs. Nevertheless, it is not possible to simple apply MANET concepts on WSNs. Due to energy consumption deliberations, WSN nodes have only a skimpy equipping (CPU, RAM, network capabilities) in contrast to MANET nodes. These limitations are, however just the challenge. Research work has shown concepts how to economize with the available energy but security aspects are mostly disregarded so far. As such, it is often possible to disturb radio communication of sensor nodes because packets are not encrypted or at least signed. Furthermore, sensor applications run with full rights on the node. A misbehaving part of the application can compromise the whole sensor operating system next to other applications on the node. Thus not only the sensor network has to be secured also the operating system on the nodes. This section presents our approach on how to secure next generation WSNs. We believe that it is possible to improve the security of sensor nodes by introducing concepts from bioinformatics as well as virtualization to WSNs.

5.3.1 New Security Concepts for Sensor Nodes

In order to secure the wireless sensor network we have to secure the nodes of the network first. As mentioned above, sensor applications are mostly unsecured and inflexible. To overcome this problem, a kind of virtualization concept for sensor nodes is needed. Virtualization has the benefit of allowing to partition applications and to have the opportunity to run different tasks separated from each other. This is a security and robustness enhancement, because parts of the application running in the virtual container cannot disturb other tasks by misbehaving. Furthermore, it is possible to have different implementations in spare and to switch to the currently best fitting implementation. For example, Figure 5-4 shows three sensor networks where each of them uses different network topologies in reference to the environment. One problem is that we need extra memory for the virtualization, but on the one hand memory chips are more and more inexpensive and on the other hand it is possible to shrink the code size by compression. The additionally needed energy to run the virtual containers as well as the management overhead should be far less than the energy consumption needed for reprogramming the whole node like in the OTAP method.

Additionally, we believe that the virtualization method is easier to secure. As virtualization also enriches the flexibility, node cooperation is the next important step. That, for example, cooperative transmissions could have energy consumption benefits as well as a kind of self-healing and therefore a security relevant role was shown by Woldegebreal et al. in [56]. When sensor nodes resp. the sensor applications cooperate, so attacks on the communication system are simpler to detect and to defeat. Several researchers have investigated concepts from the Autonomic Computing research field for sensor networks but the combination of virtualization, cooperative nodes and self-x concepts is new. As Figure 5-4 shows, each node in our concept is extended by a management layer. This layer takes over all management tasks like switching the virtualized services on or off and deciding the network topology for example. A security engine on top of the management layer is supposed to decide concerning self-healing and self-protection aspects. The security engine has therefore to inspect the operating system by processing self-tests and the outer communication by for e.g. an kind of intrusion detection. The use of feedback messages can additionally help the security engine to get a better network overview as well as helping the management layer to decide when to cooperate with other nodes. Slightly costs have to be estimated in respect of enhancing security and dependability. Security is not available free of charge, so even if the focus of this work lies on security tasks, the energy consumption aspect cannot and will not be neglected.

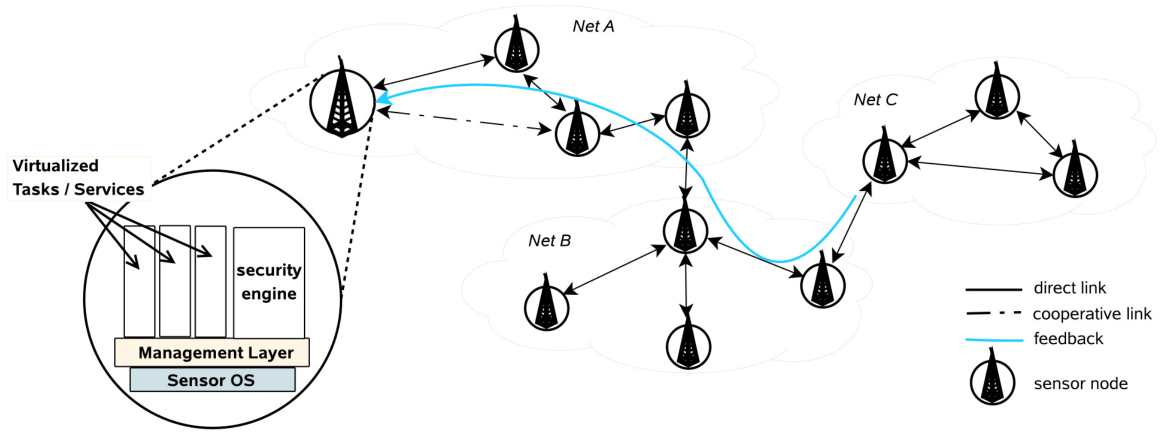


Figure 5-4. Virtualization in WSNs.

A full virtualization (node emulation) does not make sense, because it would be too complex and thus energy consuming. So, we have to investigate which degree of virtualization is possible and how a sensor node resp. the network can benefit from cooperation theoretically and practically. With this previous work in mind, further investigations will scope on how to combine bioinformatic and WSN concepts. To support the sensor operating system we have presented the idea of a management layer with a build upon security engine. These extensions make it possible to enhance the security of current sensor nodes. Next to the further development of our concept, an implementation of a secure sensor node prototype will be done. Therefore, we need a secure central storage place and a secured message bus on the node. In the next section, we present our implementation of a blackboard based, publish/subscribe architecture for WSN nodes.

Virtualization techniques in combination with self-x and trusted computing concepts are the key for building secure sensor network platforms of the next generation.

Nevertheless, solving the time and resource consumption problem of our security enhancement is an open question which we have to regard.

5.4 A Secure Blackboard-Based, Publish/Subscribe Architecture for Wireless Sensor Nodes

Sensor nodes are on the one hand small in size and economically cheap to produce, but on the other hand the applications on the nodes get more and more complex. As such, an application developer has to cope with limited resources. Memory can be efficiently used by sharing data, and the life-time of a battery can be extended, when the node has long power-saving sleep phases. This section presents a publish/subscribe architecture that achieves these two aims. We introduce a blackboard which is used for centrally storing published values, such as measured data from a monitored sensor. This makes it possible to share stored data without monitoring the sensors once again, which is advantageously concerning power consumption, memory space, and reaction time.

In addition to the proposed publish/subscribe method for sensor nodes with its notification possibilities, our architecture fulfills also real-time requirements. We show how the well-known sensor operating system MANTIS OS can be extended by a real-time enabled, blackboard-based publish/subscribe architecture. This architecture and first of all its implementation is of special interest for cross-layer optimization of sensor applications. Cross-layer approaches benefit from this architecture because the available implementation can be used as an efficient framework for central storing and managing of shared values.

5.4.1 Publish/Subscribe Paradigm

For a dynamic interaction between components, asynchronous communication models are used, like message passing, broadcast, and publish/subscribe. Eugster et al. describe in [57] that only the publish/subscriber communication fully assists all three decoupling dimensions, which are the time, space and synchronization decoupling.

Karl and Willig explain in [58] as well as K¨opke et al. in [59] that a WSN application can also benefit from the publish/subscribe paradigm. As such, before analyzing the suitability of publish/subscribe for sensor node applications, we should have a deeper look at the publish/subscribe basics first.

A subscriber is not interested in all events a publish/subscribe service may support. So the subscriber specifies the events it is interested in and cuts the amount of all possible events towards its interest. There are three different types of subscriptions possible, namely: topic-based, content-based, type-based.

At a topic-based publish/subscribe scheme, participants subscribe to a topic or subject, which is identified for e.g. by keywords. This is like joining a group, where published data for the group is forwarded to all group members. A mailing list is a good example for this type of publish/subscribe, where every topic is like an event service of its own.

In a content-based publish/subscribe scheme, the decision if a subscriber gets a notification or not, is based on present keywords in the content of the event. A subscriber defines filters or constraints, which identify valid events. If a published value fulfills the subscription filter, the notification service informs the active subscriber. The other way round, if it is not valid the subscriber gets no notification message.

The type-based publish/subscribe variant enables the subscriber to specify in which kind of values it is interested in. An event can be of different subtypes, for e.g. a number could be of the types integer or float. Type safeness may be an argument to use this type of publish/subscribe scheme. The content-based scheme is an extended form of the topic-based one. At the topic-based publish/subscribe, every publication is forwarded to every subscriber of the event. The content-based scheme uses an addition filter, allowing to only notify the subscribers of the event with a matching subscription.

5.4.2 Publish/Subscribe Architecture for Sensor Nodes

Publish/subscribe architectures for distributed systems are widely available. So if such an architecture is fitting for a sensor node, we can use the architecture for an implementation. Unfortunately, traditional publish/subscribe architectures are not suitable to sensor nodes, because they do not take into account the aforementioned resource limitations.

As K¨opke et al. describe in [59], a blackboard can be used for storing shared values at a central point. Within a data-centric control interface, setting and accessing data is possible. In their implementation for TinyOS, they use a publish/subscribe based control interface. The interesting part is how they solved the space problem which leads to the fact that concepts like the well-known Elvin concept [60] being difficult to implement in WSNs. K¨opke et al. use a blackboard for storing the data. In their implementation, each published information is defined by a unique number, a so called *notification event number*. A publisher writes its data on the blackboard and then informs the blackboard about the publication. Because of the unique event number, the publisher can tell the blackboard the corresponding event for the published value.

The blackboard itself consists of three elements: the index table, the subscriber table and the subscriber flags. The tables are of constant size, which have to be calculated at compile-time. The index table consists of two entries for each event, namely the event offset and the event count. The offset value states the relative start position of the event at the subscriber table and subscriber flags. The count value gives the amount of subscribers for the event. So the index table has a size of twice of the number of events.

The subscriber table stores the handlers for each subscriber. The position of the handler can be calculated from the offset value in the index table. An entry in the subscriber array has two flags for each subscriber. The first flag is the published flag and the second one is the subscribed flag. The published flag is set if an event is published for the subscriber. The subscriber flag is set only if the subscriber is currently subscribed to the event, otherwise the flag is clear. Subscriptions and unsubscriptions can be done at run-time, but the storage for the subscriber is reserved at compile time. The subscriber flags are stored in RAM, all other tables are transferred into the program structure at compile time. So the concept uses a minimal amount of RAM storage.

This concept however has the disadvantage of not being flexible. The number of events and subscribers, and also the handler addresses have to be known at compile time. A modification at run-time is not possible, because the storage is in ROM. So a subscriber can not declare it is interest in an event at runtime, this is done at compile time. The unsubscribe functionality is more like a pause function, where the subscriber declares its not willing to get more notifications at this point in time. The concept uses the publish/subscribe paradigm for decoupling in space and time. In K¨opkes concept subscribers can only make a topic-based subscription for an event. On every event change they get notified if their subscription is active. The subscribers are notified by calling the function whose address is stored at the handler field.

5.4.3 Static Data Storage

K¨opke's concept allows only topic-based and static subscriptions. Our idea is to extend the given concept to a content-based system, so that subscribers can set and modify a subscription filter, like type and limit, as well as the stored handler address. Instead of storing index and subscriber table in the structure, these now have to be stored in RAM too, in order to modify the values at run-time. Because a WSN node has only limited resources, especially RAM, the maximum number of events and subscribers is limited. At compile time, the required storage size has to be calculated and reserved, since most operating systems for WSN nodes have no *malloc()* functionality available (MANTIS has none).

A developer should know the number of events which are published by a publisher thread, and subscribers for the events. Subscriber and events are in a relationship to one another. Each subscriber has to know the name or id of the event it wants to subscribe to. So the storage capacity required by the concept is calculated at compile time. A test should show at compile time, if enough additional space is available to handle the storage of the blackboard.

K¨opke et al. use one published flag per subscriber in the subscriber flags array. In this concept, we use one publish flag for each event. There can be several distinct subscribers for one event, with different subscription filters. The subscriber flags array here is similar to K¨opkes architecture, within the difference that one publish flag followed by n subscriber flags and one block flag are used for one event. The block flag is needed for safeguarding reasons. To calculate the bit position of the flags, we need no additional field in the index table. The offset value used to calculate the handler reference can also be used here for the subscriber flags. If a publisher sends a new value for an event to the blackboard, the published flag for this event is set and in the notification state entered later, only the events with an activated published flag have to be processed. In contrast to K¨opke's concept, this publish/subscribe system is able to handle content-based notifications. On subscription, the subscriber commits the event identifier and the subscription filter. This filter consists of two fields, namely the subscription type and subscription limit. When a new value is published and the subscriber is subscripted for this event, the blackboard has to prove, if the subscription filter is fulfilled or not. On a positive decision, the subscriber gets a notification. As in K¨opke's concept, this could be done by invoking a handler.

The advantage of this concept over K¨opke's is that it is more flexible. A subscriber is able to unsubscribe, if a subscription is not needed anymore, and a second subscriber can later subscribe and can reuse the reserved storage from the first subscriber. In order to reduce the amount of storage, the index table can be skipped if the number of subscribers is fixed. For e.g. having fixed the number of subscriber for each event to 6, the offset in the subscriber flags is 8. So for each event, only 1 Byte has to be reserved for the flags per event statically. The problem here is fragmentation. In the worst case, assuming at least one subscriber is available, space for 5 additional subscribers are unused, but the storage for them is still allocated. If more than 6 subscribers are interested in the event, a dummy subscriber can be created, which publishes to new event on notification of this dummy subscriber. Additionally subscribers can subscribe to this event, instead of the fully subscribed root event.

5.4.4 Dynamic Data Storage

In a static storage, storage reservation is necessary at compile time. Space for events and subscribers is calculated and each gets its own part of storage space. Another concept is to reserve a given size of storage for the notification system at compile time and reserve space for events dynamically at run time. On creation of an event, a data container is created, which holds all relevant data of the event. The published values as well as the subscriber filters are outsourced and only accessible by a pointer. Therefore, the event container holds the pointer addresses to these data.

Beside the pointers, information like the number of subscribers and the number of publishable values is also stored in the container. The number values are needed to calculate the last position of either the published value array or the subscriber arrays for subscriber limits and types. The subscriber flags are stored in the event container. Unlike the static variant, the subscriber flags are not limited and can have different sizes for each event. The number of subscribers can be found in the container, so the size of the subscriber flags in bits is two plus number of subscribers. Each event container has a pointer to the next container, so each container is reachable through its predecessor. A direct access on a container could be done by using an index table, which holds all next pointers.

Storing pointers to the values instead of the values in the container has the advantage that sharing is possible. In a situation where two different events have the same subscribers, the pointers for subscriber limits and types are the same. The individual subscribers of the two events share the subscriber array and so reduce the amount of needed space. A modification of the filters will influence both events at the same time, this means that cycle counts are reduced for modifying the filters. Secondly, not only the last published value can be stored, now several values can been stored externally with the pointer to the first value in the container. Each event container can handle a different number of published values.

The containers are linked by a next pointer, so a list of containers arises. This list can represent a priority order, where a container in the front has a higher priority as its successor. The priority order can be changed at run time by swapping the next pointers. At the notification phase, the blackboard starts with highest priority containers. As in the static concept, only event container, whom published flag in the subscriber flags array is set, need further processing. If at least one of the subscriber flags is set as well and the corresponding subscriber filter is true, the blackboard marks the subscriber to be notifiable. After all containers are processed, the notify list with all marked subscribers can be processed. A method like first-come-first-serve should be adequate here, with the subscribers of higher prioritized events first. Reordering should be used for containers which are used often. Reordering can be done at run time, depending on a situation or event.

The event container is easily expandable for the developer. Additional values can be stored in the structure by adding new variables in the event container. The main problem with the dynamic storage concept is the needed storage capacity and the needed computational time for modifications. Specially on publishing a value for an event, we need a index table to get the current pointer address to the container for the event quickly. The index table is only needed for publishing. Without the table, we would have to search for the correct container by hopping from one container to the next, until the correct container is found. The index table, holding the id of the event at its pointer to the container, has also the problem of being static. Since sensor operating systems like MANTIS OS do not support *malloc()* functionality, the storage for the index table has to be reserved at compile time. So even when the dynamic storage concept is used, the amount of accepted events is limited at compile time.

5.4.5 Combination of Static and Dynamic Storage

Since neither the static nor the dynamic approach seems to really suit to WSN nodes, a combination of the two should be discussed. The idea is to reduce the used space while keeping flexibility. The static concept limits the subscriber to define better filter constraints for their subscription. A subscriber can only post one filter for each event. It should be possible for the subscriber to define several rules, which have to be valid for notification. In Elvin [60], a special subscription language was used for defining subscription filters. Defining a special subscription language is expensive regarding storage and computational time. Another idea is the creation of special subscriber type, used for dummy subscriptions. When the first subscription filter is valid, a sub-event is notified by a dummy subscriber with the second filter rule. The second stage subscription rule is validated after the notification and the real subscriber can be informed if this second subscription filter is also valid. This idea leads to a tree-like subscription filter net. So a nesting of subscriptions covers the same complexity as a subscription language, like the one in Elvin.

At the dynamic concept, an index table has to be used for finding the requested event container or the list of container has to be gone through to find the correct container. In the static variant, no index table is needed, because the number of subscribers is fixed. The idea now is to get a combination of the two concepts, but the question is how to do this. First, an observation may help. Subscriptions are for an event and the event is well known to the developer. The application developer uses a subscriber for an event with a fixed identifier. So the developer knows the total number of subscribers and events in the system. As a result, the developer is able to reserve as much space as necessary for the notification system at compile time.

Because the dynamic concept uses storage space wastefully, a more static concept is needed. Figure 5-5 shows the combination of static and dynamic storage for one event. A management frame is used instead of a container. This frame consists of six parts, which have a total size of 7Byte. The first part contains the well known subscriber flags. The number of subscribers is limited in this example to six, so the first management frame part has exactly a size of 8bit. This is profitable, because having a 8bit CPU, flags can be set and cleared rapidly without being vigilant about byte bounds.

The second part of the management frame consists of eight 1bit flags for describing type and status of the event. We will discuss the meaning of this field later in this section. The remaining parts of the management frame are the four pointers to the published data, the subscriber limit, type, and handler.

A pointer itself has normally a size of 32bit, but with the observation in mind that every data or subscriber value has a size of 32bit, only storing the position instead of the real pointer address is enough. In the management frame, each of the positions, which can be stored, has a size of 10bits. So for all four positions, we need 40bits, and with adding the 16bits from part one and two, we have 56bits, which are 7Bytes, in total for a management frame. As in the static variant, an index table is not needed, because each management frame has the same size. So calculating the start position of event x is easy.

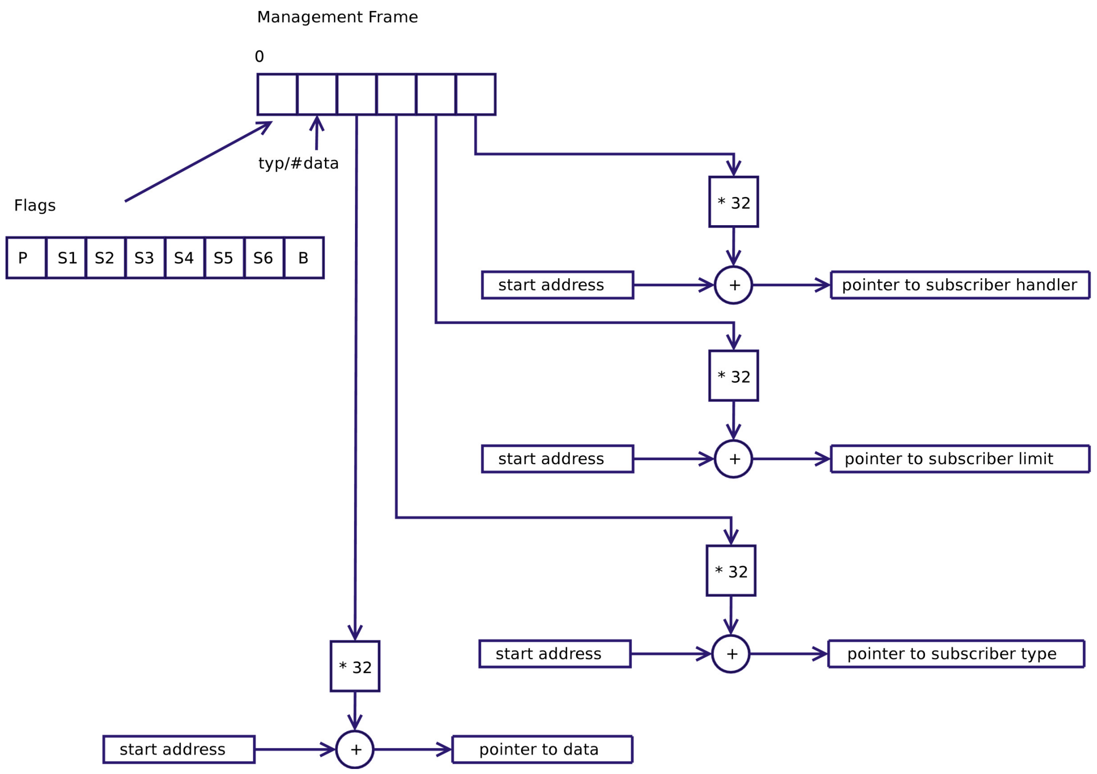

Management Frame
0
typ/#data
Flags
P
S1
S2
S3
S4
S5
S6
B
* 32
start address
+
pointer to subscriber handler
pointer to subscriber limit
pointer to subscriber type
pointer to data

Figure 5-5. Combination of static and dynamic storage.

We have reserved 10bits for storing the position, instead of the pointer. The question is know how to use this position for calculating the real pointer and is the size of 10bits huge enough? First we will show that 10bits are sufficient by taking into consideration that data is stored on a sensor node. Secondly the pointer calculation with position value as an offset value is shown.

With 10bits representing the position of a value, there are 210 possibilities. So 10bits are sufficient to number at least 1024 values. Each value has a size of 32 bits and so in total 32bits · 1024 = 32768bits or 4096Byte are addressable. A sensor node like the Mica2dot has only 4kByte RAM, the considered 10bits are adequate for addressing 32bit values. With a start address and the 10bits for the position, the pointer can be calculated. The position multiplied with 32 gives the offset, which has to be added to the start position. The result is the searched position address of the value.

When the total size of RAM would be reserved for the notification system, realistically at most 35 events can be created with 35·6 = 210 possible subscribers. When all of them store an individual subscription filter and all events store one published value, than 35·7Byte + 35·96Byte= 3605Byte are used. Not surprisingly, the amount of needed space can be cut, if subscription sharing is used, like in the dynamic variant. Additionally, unlike in the dynamic variant, the subscription blocks for all 6 subscribers can be partly shared.

As mentioned, no index table is needed, because of a static size of the management frame. Therefore, the number of possible subscriptions to one event is fixed to a number like six. If there are more than six subscribers for an event, the idea is to create a new event and make one special subscription at the first event, which is always true at examination. The new event is called a sub-event of the first one. There may be at most six sub-events for each event and also each sub-event could have its own sub-events.

So the number of subscribers for an event is only limited by the available storage space and not by the architecture. The special subscription used here needs a significant value in the subscription type and the position of the sub-event in the subscription limit field. As seen in the dynamic concept, a priority based scheme is possible. The subscribers in the root event have a higher priority than its sub-events.

Next to prioritization, it is possible to create a sequence of subscriptions, which represents multiple equations of a subscription. For example, a subscriber who would like to get a notification, provided that the published value is larger than a limit x and smaller than a limit y, defines two subscription filters. The first filter, from the type forwarding and greater with the subscriber limit value of x, only notifies a sub-event if the greater equation is valid. The subscriber handler field of this first subscriber filter holds the address of the sub-event. The second subscriber filter is the filter of a subscriber of the sub-event. This filter is a normal subscription filter with a smaller type of subscription equation and the limit y. If the first equation is valid, the sub-event is activated by a notification. If also the second subscription equation is valid, the real subscriber gets a notification. The positive effect is that the published value is stored only once.

Both management frames have the same position stored for the published value. It is also possible to extend this strategy for n equations, but than reducing the *offcuts* is more and more important. When sub-events are created, a tree of events is built. In this scenario, it is possible to use the quenching strategy from Elvin. If no subscriber of a sub-event is active, all subscriber flags are cleared, than the subscriber flag for the sub-event should also be cleared in the subscriber flags array of the root event. This has the consequence that the blackboard cuts the sub-event branch of the event tree. Quenching for sub-trees reduces computational time and should be used as often as possible.

As mentioned before two bits in the management frame are reserved for the event type. So far there is no sensible necessity for using the event type. A later possible usage may be the decision if one of the bits should be used as a storage flag. This flag than indicates if the blackboard information is stored in RAM or in the ROM part. A set storage flag lets the blackboard use for calculating the correct storage pointers the start address of the RAM part as a cleared flag means the start address of the ROM part has to be taken. If more than one publication should be stored, the difference between the number of the publish and the subscriber type pointer has to be greater than one. Than there is enough space reserved for storing more than one published value. For simplification the front value, which is in the position of the pointer address should always been the latest value. So, on each publication all published values have to be pushed 32bits to the right within their space reservation. Then the first entry is overwritten by the new published value.

5.4.6 Concept of a Real-time-Enabled Publish/Subscribe Architecture

A value is published on the blackboard when the publish function is executed within a thread. The function needs as parameters the event id, which is unique, and the new value, which should be published. When the blackboard thread is activated later, the blackboard works off the publishing procedure. First it tests if the event is blocked by another process. If the blocked flag in the subscriber flags array is set, the publisher has to wait for unblocking or skip the publishing process for this event. When the block flag is not set, the blackboard set the block flag and the publish flag. Now no other process can disturb the publishing process for this event. The publish flag is set and so leads to later verification of the subscription. If all subscriber flag are cleared the published flag should also be cleared, because when there is no subscription, no subscriber can get a notification. Also it could be possible to inform the producer about this situation. This process is called quenching and instructs the producer to skip further publishments for the event. When at least one subscription is made for the event, the producer has to be informed once again, now to stop the skipping.

In our concept, not the complete management frame has to be modified; only the first byte needs a modification. The first byte consist of the subscriber flags, which should be notified as described. In order to store the published value in the right position, the publish block of the management frame is read and the value is stored at the real address. After finalization of the publishing process the block flag in the subscriber flags array is cleared.

A consumer thread calls the subscribe function of the blackboard API for creating a subscription for an event. The event id and the subscription filter values are given to the function by parameters. The subscriber gets a unique subscription number, which is the next free number of the all subscriber flags in the event, back as a result of the function call. Like in the publishing process, the block flag is tested and set. If no other subscriber flag is set before and quenching is used, the producer has to be informed, that there is at least one subscriber, who is interested in its published value. The first two bytes of the management frame have to be modified. In the first byte, the subscriber flags array, the corresponding subscriber flags has to be set. In the second byte, the type field, also the corresponding flag is set, standing for a reservation of the subscription block for the subscriber.

Since a structure optimization for reducing *offcuts* is not done so far, the data flags block should be a copy of the subscriber flags. The needed next free subscriber number is the first position of a subscriber flag, which is cleared and which representing flag in the data block is not set. If no space is available in the management frame for an additional subscriber, a new event is created, called sub-event. One of the subscribers is copied to the new event, meaning coping subscription filter values and setting the subscriber and data flags in the new sub-event. In the gap of the old subscriber in the root event, a new subscriber is created with a subscription filter of a special forwarding type. This subscription is always true and on notification processing, the new sub-event gets activated by notification. Because the root event holds the position to the published value, the position can be copied to the position stored in the sub-event.

The subscription filter values are subscription type, limit and handler. These values are stored at the position, which stand in the three position blocks of the management frame. The first subscriber takes the first 32bit beginning at the given position, the second subscriber the next 32bits and so on. This procedure is done for all three values.

An unsubscription has to be divided in a pausing and a real unsubscription request. Pausing means that, the subscriber wants to skip notifications for a short time and wants to resume later. Unsubscribing means that the subscriber gives its subscription up and that the reserved memory space can be used be another subscriber. The subscriber executes the unsubscribe routine with the parameter of the event id, the subscriber number and if its willing to pause or not. Like in the other processes, the block flag is tested and set first. If the corresponding subscriber flag is set, the flag will be cleared. If not pausing, but unsubscribing is chosen, also the corresponding data flag is cleared. After all, the block flag is cleared and the unsubscribe process has finished.

If a new value is published and a subscription inequality of at least one subscriber is fulfilled, the subscriber needs a notification. But how to notify the subscriber? Before possible solutions can been discussed, the goals should be clear. A notification method should:

- be effective concerning space and time
- be implementable (Mantis prototype)
- fulfil the three publish/subscribe constraints

Subscribers can be splitted into two parties. Subscribers, which are interested only in the notification itself, not in the published value, are in the first party. These subscribers are namely the alert subscribers. The second party of subscribers need the value for their further proceeding, like sending a message with the published value to another node. These subscribers react on a notification like the alert ones, but also need the value at notification. Each of these subscribers could be build as an alert subscriber, which requests the value at notification or the value is passed to the subscriber with the notification.

Subscribers who are only interested in getting a notification as an alert have three choices in getting a notification. First, the subscribers could poll for a notification. The blackboard needs a notification queue, in which all subscriber ids are stored which have to be to notified. On a poll request, the blackboard will give a true back as response, when there is an entry for the subscriber in the queue and false otherwise.

After every valid polling request, the found subscriber entry is removed form the queue. The problem with the polling method is, that when its used too often, too much computational time is wasted, having an active consumer thread. So the thread with the waiting and polling subscriber disables the operating system to switch to lower electric consumption state. The time the subscriber waits for the notification may be used to let the micro-controller sleep, but with an active polling process this not possible. In a situation, where the producer of the event is not in the same thread of the subscriber, it is possible to reduce the polling to a minimum. On a reactivation of the thread, only one polling is allowed, because a second request would always return false. But even in this situation, the polling method inhibits an efficient power management. The only positive fact is, that the subscriber defines the point in time, when the notification should be processed.

A second idea is the usage of a function call. In some operating systems a alert function is available, which starts a function, when a given point in time is reached. For execution of the function, a reference to the function is given to the alert process on creation of the alert. The same could be done here. The function would be executed in the context and time of the blackboard. So the subscription thread is definitely not blocked and does not wait or pull for getting a notification. The problem is, that the function may use some variables from the subscribing thread. This is not allowed, better not possible, when the function is executed at the blackboard thread. Only global variables are reachable for the function in order to publish the computed new value. Another idea may be, that the function uses the publish function to make the value known. But then the whole process chain has to be build as nested functions. This complicates the process unnecessary in context of debugging and implementation interested.

The second point is that this function call method breaks the multi-thread concept. The process chain would be computed at the blackboard thread. The main scheduler would be useless in this situation. So a new additional, inline scheduler must be implemented in the blackboard. This scheduler schedules all function calls for the blackboard. As we can see, the function call method is tricky and so arduously to proper implement.

A third idea, how to implement a notification, is the idea of a locked subscriber block in the subscribing thread. At the first sight, blocking the subscriber is not wished, concerning the decoupling rules. But here, not the whole subscriber is blocked, only a part of it, the part which handles the things to do when a notification arrives, is locked. Instead of out-sourcing the code in a function, which has to be executed on notification, here the code is still in the thread, but locked with a global available semaphore, which the subscriber has defined. The previous discussed notification model with function calls can been transformed in this model. The code of the notification functions for the subscribers build subscription blocks in a special subscription thread. Each thread is locked by a semaphore, which is unlocked if the blackboard thread calls the notification for the thread. Instead of executing functions at notification, the semaphore is unlocked and the subscription thread is activated once.

The only thing the blackboard has to do, is to unlock the semaphore. In the subscriber block of the thread, it has than to be decided, if the semaphore should be relocked again or not. The positive effect on this method is that first the subscriber decides the point in time of the notification in the thread and secondly thread variables are still available. The problem is, that there could be a problem with the scheduler, when the subscribing thread is not activated contemporary.

When the blackboard clears a semaphore for a subscriber, the subscriber could execute the now unlocked code. In a situation, where a producer publishes a new value for this event before the subscriber thread was activated, the blackboard once again tries to notify the subscriber by unlocking the already unlocked semaphore and the subscriber only gets the notification for the last published value. So when a notification guarantee should be given, than the scheduler has to prior the thread with an outstanding notification. The needed modification for the scheduler should be minimal. An additional queue is needed, where the blackboard writes all subscriber ids to notify down. The scheduler works off this list and gives the corresponding subscriber thread the highest priority. In the next step, the subscriber id is removed form the list, so that on the next scheduler round, the thread gets its old priority back.

So far, we have shown how to store events and subscriptions on the blackboard and how producer and consumer can use the blackboard API. Now we will discuss the blackboard process, which is settled in an own thread, the blackboard thread. The assignment of the blackboard thread is to process all publish and subscribe requests as well as verifying the subscriptions and start the notification process if necessary. The blackboard thread can join five different states. These are: *sleep*, *modify*, *prove*, *notify* and *optimize* states.

The *sleep* state is always the first and last state in the blackboard process. First, being in this state, it is proved if at least one producer has published a value. This can be done, by looking in the publisher queue. If this list has at least one entry, the blackboard thread switches to the modify state, else the thread is let sleep and the scheduler can switch to another waiting thread. The publishing queue is needed, because the publish/subscribe paradigm demands a decoupling in time. When a publisher would access the storage management directly, then one part of the blackboard assignment would be done by the producer.

In the *modify* state, each entry in the queue is worked off in a first-come-first-serve manner. This is done because of the fact that a subscriber should only get a notification for an event, which is published after the subscription. For the unsubscribe request its vice versa, if a consumer unsubscribes no further notification should arrive after the unsubscription. If all subscriptions in the queue would have to be done before publishing in the blackboard, then the resulting process order may be incorrect. After the processing of each queue entry, the item can be purged from the queue. When the queue is empty and at least one publish entry was processed, than the thread switches to the prove state. Else the blackboard has finished and switches to the sleep state.

In the *prove* state, the blackboard process goes over all management frames, in order to find an activated publish flag in the subscriber flags array of the frame. If one is found, it is proved if one or more subscriptions are fulfilled. For each of them an entry with the subscriber id and the subscription handler address is stored in the notification queue. Next, the publish flag is cleared and the next management frames are tested. When all frames are worked off and at least one notification has to be done, the blackboard thread switches to the *notify* state. Else, no notification has to be done and so the *sleep* state is activated.

In the *notify* state, all notification assignments, which are stored in notification queue, have to be worked off in a first-come-first-serve order. Therefore, all notifications semaphores, which reference is the subscription handler address, are cleared. Following, the value of the thread id, which belongs to the subscriber, is pushed to the scheduler. As described in the section before, the scheduler decides, when a thread is activated. When the queue is processed, the blackboard finishes its process by switching to the sleep state.

An optional *optimize* state can be inserted after the publish state. The assignment of the optimize state is to reduce offcuts in the blackboard storage. This state increases the needed computational time of the blackboard, so it should only be used, if the fixed available memory for the blackboard is nearly reached.

5.5 Policy-driven Management of Handheld Devices

In addition to wireless sensor networks, the AMICAD activity investigated the management of handheld devices. The increasing complexities of today's networks pose significant challenges to network management models. Policy Based Management is considered as a promising solution for the complex task of managing different aspects of the network, providing the means towards the effort of simplifying and automating the administration process. A policy, the basic building block of the policy-based system, is seen as way to guide the behaviour of a network or distributed system through high-level declarative directives. Policy rules constitute interpreted logic, which facilitate flexibility and adaptability in the sense that policies can be dynamically changed without changing the implementation. Traditionally, the use of policies and the development of policy-based toolkits focused on the management of fixed networks but recent efforts have been investigating policies for managing wireless ad-hoc networks of small devices. The constraints imposed by devices such as Internet tablets and smart phones forming such networks influence the design and development of policy toolkits.

This section investigates the use of Ponder2 [45], a policy framework specifically designed for limited capability devices, in managing Internet tablets. More specifically, the work focuses on the implementation and deployment of configuration management policies for these devices. This section describes the relevant tools used and the development as well as execution of the policies.

5.5.1 Development Environment and Tools

The wireless device on which Ponder2 was installed and experimented with was the N800 from Nokia. The *Maemo* platform is the core software stack for the device. It consists of the Maemo OS (OS2007 [46]) and the Maemo Software Development Kit (SDK). The platform is mainly based on open source code and its development has been carried out in collaboration with other open source projects such as the Linux kernel [47] and Debian [48]. It is based on the Debian GNU and uses the cross-platform widget toolkit based application framework such as *Hildon* as the graphical interface. The Maemo SDK provides a *sandbox* environment where the development takes place. This is based on GNU/Linux desktop system that is built on the *Scratchbox* compilation toolkit [50]. This is used to develop and port applications to the Maemo platform using another machine; Maemo 3.2 SDK [49] was used for this work, which supports the development of software for the N800.

The wireless device does not provide native support for Java. Since this is a requirement of Ponder2, we have used the JamVM [51] Java Virtual Machine (JVM). This is a new virtual machine conforming to the JVM specification version 2. It has been developed to be extremely small compared to other virtual machines and has been designed to use the GNU Classpath [53] Java class library. JamVm version 1.4.3, which has a size of 400KB-500KB, has been used in this work.

Before installing Ponder2 on the wireless device the Retrotranslator [52] has to be used to generate Java 1.4 classes. Retrotranslator is a tool that makes Java applications compatible with Java 1.3 and 1.4 JDK. It includes all language features of Java 5.0 and a significant portion of the Java 5.0 API on J2SE 1.3 and J2SE 1.4. Its use is essential because Ponder2 only supports Java 1.5 or above, whereas JamVM runs on a Java 1.4 compliant virtual machine

5.5.2 Policy-driven Configuration

The design of the Ponder2 system is simple, scalable, self-contained and flexible, and has been developed to satisfy the constraints imposed by small devices. Ponder2 supports two types of policies: authorizations and obligations. The former allow or deny requests from one managed object to another, whereas the latter are event-condition-action rules. A number of obligation policies have been implemented for the purpose of this work, which have been used to configure the system settings of the N800 Internet tablet. These policies include rules that control the wireless connectivity of the device (Bluetooth, WLAN), and for setting different levels of volume and brightness on the sound card and the screen, respectively.

5.5.2.1 Utility Scripts

Here, we describe the implementation of a policy that adapts the screen brightness based on the battery level. The objective of this policy is to conserve the energy of the device and extend its lifetime when away from an energy source. In order to realize this policy, two scripts need to be implemented:

- ScreenUtility script, which can be used to obtain the current screen brightness level of the device and allows the brightness level to be adjusted with *gconftool* [54]. The relevant script is shown in Figure 5-6.
- BatteryUtility script, which can be used to obtain the battery status, i.e. charging or not charging, and also the power level, using *dbus* [55].

```
NewDisplayBrightness=${1}
MinDisplayBrightnessLevel=1
MaxDisplayBrightnessLevel=`gconftool-2 -g /system/osso/dsm/display/max_display_brightness_levels`
CurrentDisplayBrightness=`gconftool-2 -g /system/osso/dsm/display/display_brightness`
if [ "$NewDisplayBrightness" == "" ]; then
        NewDisplayBrightness=$CurrentDisplayBrightness
elif [ "$(echo $NewDisplayBrightness | grep "^[0-9]\{1,9\}$")" != "" ]; then
        if [ $NewDisplayBrightness -ge $MinDisplayBrightnessLevel -a $NewDisplayBrightness -le
          $MaxDisplayBrightnessLevel ]; then
                exec `gconftool-2 -s /system/osso/dsm/display/display_brightness
                    $NewDisplayBrightness -t int`
        else
                echo "Warning! invalid brightness level, please enter a valid value between
                   "$MinDisplayBrightnessLevel" to "$MaxDisplayBrightnessLevel"."
                exit 1
        fi
else
        echo "Warning! invalid argument, please enter a valid argument for example 'ScreenUtility
            ["$MinDisplayBrightnessLevel-$MaxDisplayBrightnessLevel"]'."
        exit 1
fi
echo "brightness:"$NewDisplayBrightness
exit 0
```

Figure 5-6. ScreenUtility script.

5.5.2.2 Managed Objects

In addition to the scripts, three managed objects (MOs) need to be implemented: a Screen MO, a Battery MO, and a BatteryMonitor MO. The first executes the ScreenUtility script and, as shown in Figure 5-7, implements methods to obtain and set the screen brightness level of the device using PonderTalk commands. The Battery MO executes the BatteryUtility script and implements methods that obtain the battery charging status and power level (Figure 5-7).

The BatteryMonitor managed object contains and controls a Battery timer – a thread which has been used for creating and sending events at a specified frequency. The timer uses the Battery MO to obtain the battery status and power level and passes them via the events. It also keeps a list of events in the event list. Figure 5-8 depicts the methods of this MO which have been implemented as PonderTalk commands.

```
public class Screen implements ManagedObject {
        @Ponder2op("getBrightness")
        public int getBrightness() {
                this.excuteCommand("");
                return m_brightness;
        }
        @Ponder2op("setBrightness:")
        public void setBrightness(int brightness) {
                this.excuteCommand(String.valueOf(brightness));
        }
}
public class Battery implements ManagedObject {
        @Ponder2op("getLevel")
        public BigDecimal getLevel() {
                this.excuteCommand();
                return m_level;
        }
        @Ponder2op("isCharging")
        public boolean isCharging() {
                this.excuteCommand();
                return m_charging;
        }
}
```

Figure 5-7. Screen and Battery managed objects.

```
public class BatteryMonitor implements ManagedObject {
        @Ponder2op("addEvent:")
        public void addEvent(P2Object event) {
                m_timer.addEvent(event);
        }
        @Ponder2op("cancel")
        public void cancel() {
                m_timer.cancel();
        }
        @Ponder2op("getDuration")
        public int getDuration() {
                return m_timer.getDuration();
        }
        @Ponder2op("removeEvent:")
        public void removeEvent(P2Object event) {
                m_timer.removeEvent(event);
        }
        @Ponder2op("setDuration:")
        public void setDuration(int seconds) {
                m_timer.setDuration(seconds);
        }
        @Ponder2op("start")
        public void start() {
                m_timer.start();
        }
}
```

Figure 5-8. BatteryMonitor managed objects.

5.5.2.3 Policy Implementation

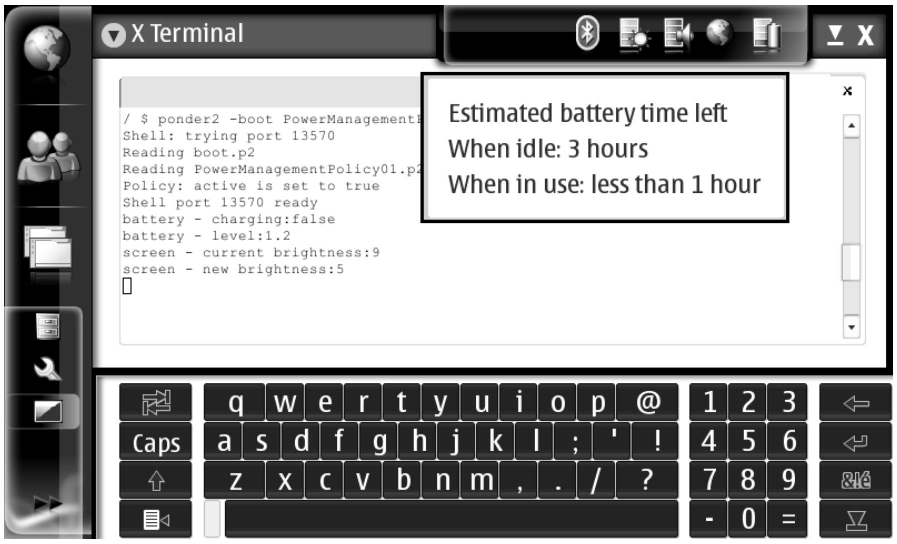
Based on the managed objects described above, a power management policy can be implemented in Ponder2 as shown in Figure 5-9. This loads the BatteryMonitor and Screen managed objects in the root domain. It also creates an event template and stores it in the root/event domain. The Battery event template contains two parameters – charging status and power level – which have been previously added to the BatteryMonitor MO's event list. The battery event is assigned as the policy event and the condition of the power management policy specifies that if the battery is not charging and the power level is less than or equal to 50%, then the policy action is enforced. The latter sets the screen brightness to 50% (level 5). It should be noted that the frequency at which battery events are received by the policy engine is set to 60 seconds. Figure 5-10 demonstrates the result of the policy execution on the wireless device.

```
root/factory at: "batterymonitor" put: (root load: "BatteryMonitor").
root/factory at: "screen" put: (root load: "Screen").
root at: "batterymonitor" put: (root/factory/batterymonitor create).
root at: "screen" put: (root/factory/screen create).

batteryevent :=  root/factory/event create: #("charging" "level").
root/event at: "batteryevent" put: batteryevent.
root/batterymonitor addEvent: root/event/batteryevent.

pmpolicy event: root/event/batteryevent.
pmpolicy condition: [ :charging :level | charging not & (level <= 50) ].
pmpolicy action: [ :charging :level |
                        root print: "battery - charging:" + charging.
                        root print: "battery - level:" + level.
                        brightness := root/screen getBrightness.
                        root print: "screen - current brightness:" + brightness.
                        root/screen setBrightness: 5.
                        brightness := root/screen getBrightness.
                        root print: "screen - new brightness:" + brightness ].
pmpolicy active: true.
root/batterymonitor setDuration: 60.
root/batterymonitor start.
```

Figure 5-9. Power management policy.



```
/ $ ponder2 -boot PowerManagement
Shell: trying port 13570
Reading boot.p2
Reading PowerManagementPolicy01.p2
Policy: active is set to true
Shell port 13570 ready
battery - charging:false
battery - level:1.2
screen - current brightness:9
screen - new brightness:5
```

Estimated battery time left
When idle: 3 hours
When in use: less than 1 hour

Figure 5-10. Power management policy enforcement.

5.6 Conclusions

The first part of this section, presented a classification of faults and models that are observed on sensing devices in a WSN. We studied their impact and detection mechanisms. Furthermore we presented policy-based middleware approach that facilitates adaptation inside WSNs and we intend to use as the basis of a fault detection and self-healing platform. Future plans in this work involve extending self-healing approaches with a richer set of sensing modalities, where nodes prove to collect overlapping information exposing explicit or implicit redundancy in systems. We also want to further expand the expressiveness of policies for motes to provide a versatile framework for controlling WSN applications by combining low level components and primitives.

Data sharing schemes and power-saving sleep phases in wireless sensor networks can result in efficient memory usage and in extended life-time of sensor nodes. This section presented a publish/subscribe architecture that achieves these two objectives. The approach is based on the notion of a blackboard which is used for centrally storing published values, such as measured data from a monitored sensor. It is thus possible to share stored data without monitoring the sensors once again, which is advantageous concerning power consumption, memory space, and reaction time. Furthermore, we have shown how the well-known sensor operating system MANTIS OS can be extended by a real-time enabled, blackboard-based publish/subscribe architecture, which can be of special interest for cross-layer optimization of sensor applications.

Last but not least, this section reported on the use of the Ponder2 policy framework in managing devices with limited capabilities. The various tools required for the installation of the platform have been described and examples from the development and enforcement of a power management policy have been presented.

6 Distributed and Autonomic Real-Time Traffic Analysis (DATTA3)

6.1 Introduction

During the last phase of the DATTA3 activity the two main areas which were investigated were 1) how to distribute traffic monitoring data for distributed processing of such data, and 2) how well can traditional centralized relational databases cope with high amounts of traffic data. In addition, a separate investigation of how SNMP traces are actually used in real networks was performed. The three different tasks are described in the following sections.

6.2 IP Traffic Data Distribution

IP network traffic observed in the backbones of large network operators is often metered and the result of the metering process is sent to an analysis application for different purposes: *e.g.*, to charge clients for their traffic, to identify malicious network activity, to detect network anomalies (e.g., congestion, broken links), or to measure Quality-of-Service (QoS) parameters, such as throughput, delay, or jitter.

During the last decade network traffic flowing in the operators' backbone networks experienced yearly increases of 50%-100% of traffic volume [66]. A newer study on the evolution of IP traffic [64] shows that this behaviour will most probably increase even further and by 2012 the Internet traffic will be about 75 times the amount as of 2002.

Although the CPU (Central Processing Unit) performance and memory access speeds improved significantly during the same period, the performance increase of these happened at a lower rate (e.g., Dynamic Random Access Memory (DRAM) access speed improves 7-9% every year) compared to the increase of network traffic [66]. Such an unbalanced evolution makes traffic analysis even more difficult every year, as less time is available to process a single metering record. In order to address this problem, a reduction of the amount of data that needs to be processed is achieved by sampling. However, sampling has its own disadvantages. Since some data remains unprocessed, some traffic analysis applications, such as Intrusion Detection Systems (IDS) or charging may loose their required level of accuracy, as shown in [62].

Distributed systems are one way of addressing complex problems by splitting a job in multiple tasks and assigning each task to a separate processing node. Distributed analysis approaches have already shown that by sharing the workload between several analysis nodes, more analysis data can be processed per unit of time. The major drawbacks of existing distributed solutions include a reduced scalability and an application-oriented approach. The reduced scalability of those approaches does not only refer to design concepts (some use a single node that distributes the data to other existing nodes, hence being a possible bottleneck), but also to approaches that show a scalable design. However, in practice this would not be very user-friendly in an operational environment, when the number of nodes increases (e.g., they would require the reconfiguration of each node or replacement of physical hardware in case hardware traffic splitters are used). The second drawback refers to the fact that each existing distribution solution was designed for the purpose of serving a very specific analysis application only.

The architecture presented here tackles these problems and addresses both of them in an integrated solution for traffic analysis that is scalable, flexible, and standards-conforming be used for more than one analysis application.

Thus, the SCRIPT (Scalable real-time IP Flow Record Analysis) approach developed and presented further, defines a framework for building a distribution overlay for IP Flow Information Export (IPFIX) records for the purpose of distributed network traffic analysis. This solution overcomes those drawbacks observed with other approaches and provides an applicable framework to deploy traffic analysis applications in a distributed environment. In addition, due to the usage of the IPFIX protocol, SCRIPT can be used to distribute any kind of metered data as long as it can be transported in IPFIX payloads, since the operation of SCRIPT is not bound to a particular template.

6.2.1 Scenarios

In order to motivate the need for a distributed traffic analysis tomorrow a set of selected scenarios are described and respective shortcomings of centralized traffic analysis applied in these scenarios are highlighted. While Scenario 1 assumes an application which stores IPFIX records and provides a query language for those records, Scenario 2 introduces an application of delay measurement based on IPFIX records, Scenario 3 proposes an application of asymmetric-route detection based on correlation of IPFIX records belonging to the two flows of a *biflow* [69].

6.2.1.1 Data Retention

Different legal regulations require that network operators keep traffic traces for some period of time. Even without such requirements network operators keep traces for a while in order to inspect them and detect possible anomalies in the traffic observed. A centralized approach of storing traffic traces may become a bottleneck by overloading the network link to this central repository or by sending traffic traces at a higher rate than the maximum rate at which the repository can write these traces into persistent storage. Distributing this process does distribute the network and storage load to several nodes. In addition, storing all these data at a single location means that, if the repository shows a failure, all traffic traces become unavailable and no new trace is being saved. In such a case a distributed system still enables access to all traffic traces except those ones stored on the damaged node. In case of inspecting traffic traces stored a distributed system does help by running this process in parallel on multiple nodes, making results available even faster.

6.2.1.2 Delay Measurements

A delay measurement application measures the time a particular packet spent between two observation points. The PSAMP working group proposed a measurement delay application [70] using the IPFIX protocol for transporting metered data from each observation point. For each incoming IPFIX record the delay measurement application needs to lookup, if for the respective packet other measurements from other observation domains may be in place. A high packet rate has two effects on this application: there will be more IPFIX records to be kept in main memory, which increases the lookup time, but at the same time there will be less time available to process a single IPFIX record. Distributing this application will both, decrease the lookup time by storing less records in the main memory, and increase the available time to process a single record, by splitting records between multiple nodes.

6.2.1.3 *Real-time Asymmetric Route Detection*

Network operators often want to avoid asymmetric routes in their networks as these are usually caused by network problems such as congestion or misconfiguration. Routes are asymmetric, if a flow does not traverse the same routers in one direction as in the other. To detect asymmetric routes flow records can be used by examining flow records belonging to a flow and its reverse flow, whether the same routers exported these records in one direction as in the other. To be able to do this, records belonging to a flow and its reverse flow have to arrive at a single collector from all possible exporters. Similar to those scenarios above, in case of a centralized solution the central collector has to deal with high IPFIX record rates (received from all exporters) and has less time to process a single record. Distributing this application reduces the load on a single collector and increases the time available to process a single record. It is important to note that the distribution scheme has to ensure that records of a flow and records of its reverse flow arrive at the same collector.

6.2.1.4 *Shortcomings of Centralized Solutions*

Most of those key disadvantages of centralized solutions observed in all scenarios investigated can be summarized as different bottlenecks due to:

a) Incoming IPFIX data arrives at a rate higher than the maximum write rate of the hard disk or storage device;

b) The network link bandwidth of the centralized collector is not sufficient for aggregated IPFIX streams from all exporters; and

c) In case of real-time processing needs, required at collector's side, processing time of an IPFIX record will be higher than the inter-arrival time of IPFIX records.

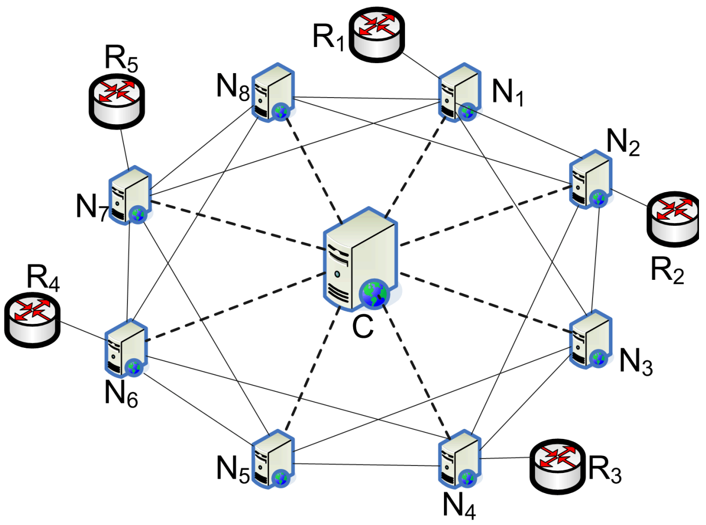


Figure 6-1. Network architecture.

6.2.2 Framework Design

The framework consists out of the network architecture applied, the Central Configuration Repository, the SCRIPT Node, and the handling of IPFIX records under investigation. These major components, their interactions, and key design decisions are outlined below.

6.2.2.1 Network Architecture

The SCRIPT flow record distribution network is organized as a Kademlia-based P2P overlay [67] as shown in Figure 6-1. Routers (R1 - R5) capture network traffic and export IPFIX records. In the following IPFIX will be used in the description; however SCRIPT also supports the NetFlow version 9 protocol, so it can be used also with exporters that do not yet fully support the IPFIX protocol. SCRIPT nodes (N1 - N8) build a P2P overlay network, they receive flow records from routers, and they distribute these records in the overlay. Traffic analysis applications (e.g., delay measurement or record storage) are running on SCRIPT nodes. Each node has two tasks: (1) to forward incoming IPFIX records to appropriate nodes and (2) to deliver a subset of the incoming IPFIX records to one or more analysis applications running on that node. The Central Configuration Repository (CCR) is involved in the bootstrap process as well as in the management of different configuration aspects of the nodes, but it is not involved in forwarding flow records.

As Figure 6-1 depicts, each router can choose any node of the P2P overlay to forward its IPFIX records. Starting at that node, a flow record routing process will start that will assure that the intended node to process an IPFIX record will receive that record. If a node is the final destination of a particular IPFIX record, it delivers the respective record to one or more analysis applications running on that node.

6.2.2.2 Central Configuration Repository (CCR)

The CCR is responsible for the node management, to support the bootstrap process of nodes, and to manage flow templates and their mapping to analysis applications.

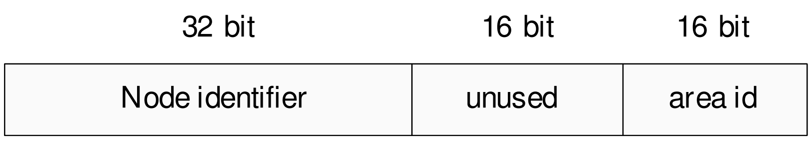


Figure 6-2. Node identity.

6.2.2.2.1 Node Bootstrap

Each node has a 64-bit identity, as shown in Figure 6-2, assigned by the CCR during the bootstrap process. The identity consists of a 32 bit node Identifier (ID), 16 unused zero bits and a 16 bit area ID. The CCR — by knowing all participating nodes due to its involvement in the bootstrap process for every node — assigns a node ID following a uniform distribution. This will result in a nearly equal number of flow records received by each node. For example, for the scenario depicted in Figure 6-1 these eight nodes will see node identifiers starting with the following byte 1F, 3F, 5F, 7F, 9F, BF, DF, and FF.

The area ID of a node identity is used to organize nodes according to geographic location in order to optimize the overlay routing, or to build logical overlays. The usage of area id is optional and is intended for extensions of the prototype.

6.2.2.2.2 Peer Awareness

The CCR also monitors all nodes and periodically checks, if nodes are alive. Whenever a node is detected as being unavailable, the CCR informs all other nodes about the change. Thus, the unavailable node will be removed from the overlay and no flow records will be sent to it any more. An additional functionality of the CCR is the distribution of application-specific messages to applications running on specific nodes, or to all application instances on all nodes. For example, such messages are queries sent by a network administrator to the flow storage application. The query is received by the CCR which then sends it to all participating nodes subsequently.

6.2.2.2.3 Template Management

The CCR stores flow templates and their mapping to analysis applications. One problem identified when dealing with IPFIX records exported by different exporters was that the same template definition received different template identifiers on those exporters. In order to address this problem, SCRIPT uses the concept of a Global Template ID (GTID). Each SCRIPT node maintains a mapping between the pair (exporter ID, template ID) and GTID. At the entry point in the SCRIPT network, the template ID is changed to GTID for each IPFIX record. Thus, two IPFIX records having the same template definition and exported by different exporters will always have the same GTID, although the template IDs that these exporters used may have been different. Each node can detect, whether the value in a template ID field is a GTID by looking at the first bit of that value. If the first bit is “1”, the value represents a GTID; otherwise it is a template ID set by an exporter, so it needs to be changed.

6.2.2.3 SCRIPT Node

The SCRIPT node represents the key component of the architecture (cf. Figure 6-1) and represents a computing device participating in the distributed traffic analysis network. In Figure 6-1 nodes N1-N8 are SCRIPT nodes. A SCRIPT node is responsible for flow record routing and for delivery of IPFIX records to analysis applications.

6.2.2.3.1 Flow Record Routing

Forwarding of IPFIX records in SCRIPT is done using a routing function. Analysis applications may have different requirements with respect to how IPFIX records are routed. For example, a delay measurement application requires that all data exported for the same packet at different observation points is forwarded to the same node, while a traffic matrix calculation application may require that all records corresponding to the same (source, destination) pair are forwarded to the same node.

Therefore, the routing function is a hash function applied to some of those fields of a flow record: hash(f(record fields)), where f() is a function that enables operations on the record fields before calculating the hash value. For example f() can be a logical AND operation on the source and destination address. The result of the routing function applied is a 32-bit identifier based on which the node, responsible for processing of that record, can be found. Based on this 32 bit identifier, the next hop of the IPFIX record is calculated using the Kademlia protocol [67]. If a next hop cannot be found, the IPFIX record is processed locally. The routing identifier is included in every flow record in a 64 bit field called routing hash ID as shown in Figure 6-3. Besides the routing identifier, the routing hash ID field contains 8 bits that are used to create temporary routing hash IDs. 8 bits are unused, while the last 16 bits may be set to an area identifier, which will cause the flow record of being routed only to SCRIPT nodes in that area (for example due to privacy issues).

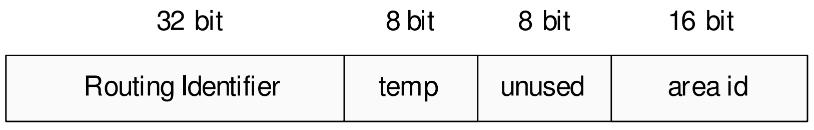


Figure 6-3. Routing hash ID.

6.2.2.3.2 Support for Analysis Applications

In order to deploy an analysis application in SCRIPT, an application ID (AppID) is chosen for it and the template (or templates) for the IPFIX records that will feed this application need to be known in advance. Respective templates are configured in the CCR and are mapped to the AppID chosen. In addition, for each newly defined template, a routing function needs to be specified. Whenever an IPFIX record is received by a node, the routing function specified for the respective template is used. If the record has to be processed locally, based on the template of the record, the AppID for those applications that use that template are retrieved and a copy of the record is delivered to each of those analysis applications.

6.2.2.3.3 Node Architecture and Functionality

The SCRIPT node architecture (cf. Figure 6-4) consists out of three main blocks: Management, Routing, and Flow Processing.

The Management block consists out of a Control Messaging component, which handles all communications of a node, a P2P Management component, which handles joining and leaving of nodes, and a Controller Unit, which orchestrates the operation of all components of a SCRIPT node. In addition, it defines an Application Programming Interface (API), which allows applications to be built on top of SCRIPT.

The Routing block includes an IPFIX Collector, which handles the receipt of incoming IPFIX records, a Flow Records Router that is responsible for finding the next hop of an IPFIX record, and an IPFIX Exporter component that is used to send IPFIX records to other nodes.

Once an IPFIX packet is received by the IPFIX Collector component, respective IPFIX records are decapsulated and passed to the Identification component. For each record, the Identification component checks, if the template ID represents a GTID. If so, the record is passed directly to the Routing and Filtering component. If the template ID is not a GTID the Identification component checks, if a mapping of (template ID, exporter) pair to a GTID exists. If there is no such mapping, a request for such a mapping to the CCR is made using the Control Messaging component. If such a mapping does not exist on the CCR either, the IPFIX record is dropped as well as all future records having that template ID, until an IPFIX record with the template definition for that template ID is received. When such a new template definition is received, it is forwarded to the CCR which returns a new GTID to be used for it and a routing function to be used with that template. Additionally, the CCR stores the new template ID and GTID in its mapping table. The final task of the Identification component, in case of IPFIX records with template IDs set by exporters, is to change these IDs with the corresponding GTID and set an internal flag (FTC) for that record, specifying that this change was just performed locally.

Once an IPFIX record arrives at the Routing and Filtering component, the FTC flag is checked. If it is set, a new 64 bit field is added to the IPFIX record, representing a routing identifier (RID) and containing a value calculated by applying the corresponding routing function to that IPFIX record. This identifier will be used by all further SCRIPT nodes to route the IPFIX record. If FTC is not set, the RID is not calculated, but read from the IPFIX record.

Based on RID and the P2P routing information, the next hop node is selected. If no next hop is found, this record is delivered to the local Flow Processing block. If a better candidate than the local node is found, the IPFIX record is marked to be delivered to that node and is put in the outgoing Queue by the Dispatching component. The IPFIX Exporter periodically reads all Queues and sends records to the next hop nodes.

The Flow Processing block includes a Pre-Processing Unit (PPU), which dispatches each record that has to be locally processed to analysis applications expecting that record. When an IPFIX record arrives, the Flow Identity Unit (FIU) identifies these applications, which require the respective record, based on the template ID of the record, and the FIU passes the record to the Flow Processor, which notifies those applications by sending a copy of the new record. The Temporary Flow DB is a special application.

Finally, the external SCRIPT application receives flow records from the Controller Unit via the SCRIPT API.

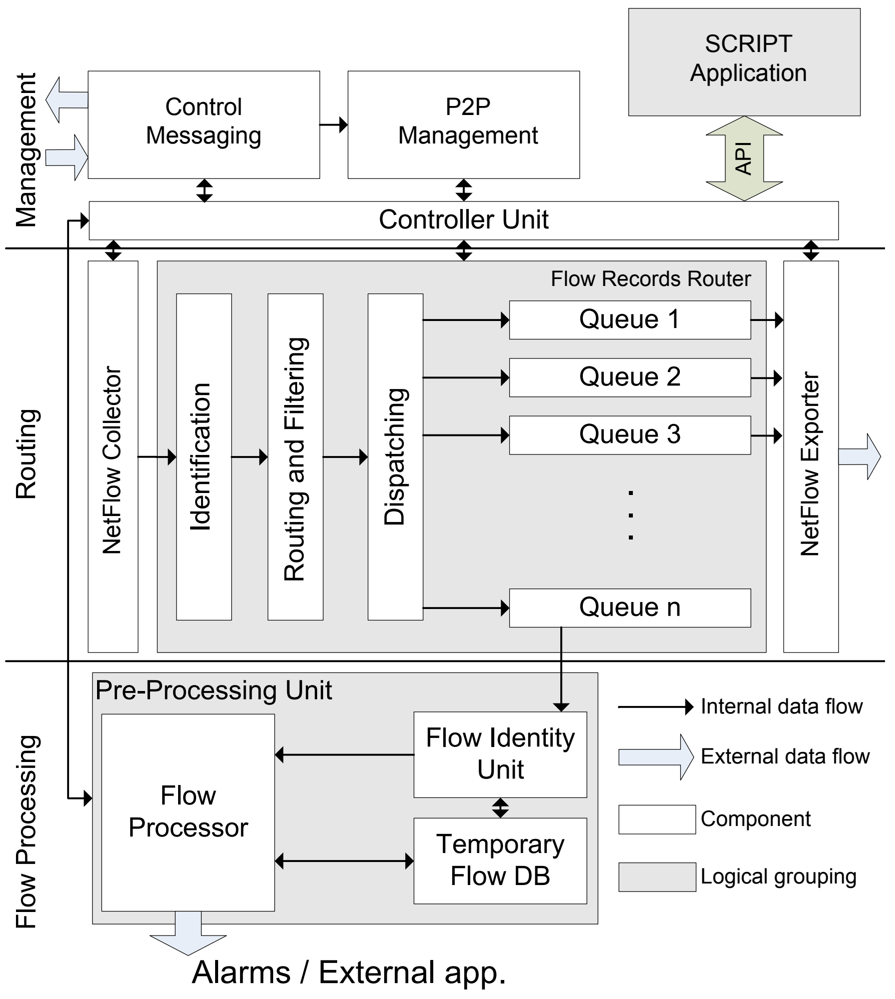


Figure 6-4. SCRIPT node architecture.

6.2.2.4 Temporary Handling of IPFIX Records

Due to a number of reasons (such as loss of connectivity, overload, or network congestion) a SCRIPT node may become unavailable for some time. Such situations may be detected by nodes connected to the node experiencing problems. In these cases, IPFIX records that have to be routed to such a node are temporarily stored by other nodes, which try after some time to deliver those records. In order not to overload a single node with all extra workload the following mechanism is applied: when a node has to forward a record to a node, which is known as being temporarily unavailable: the first 8 bits of the routing hash ID are changed and the record is re-routed using the new temporary routing hash ID. Due to the change of routing information, the node responsible for processing that record is also changed. The format of the routing hash ID contains all information required to identify, if a record has the original routing hash ID or a temporary one. It also contains the information to reconstruct the original routing hash ID when required. As soon as the IPFIX record arrives at the node responsible, showing the temporary routing hash ID, the record is placed in the Temporary Flow DB. After a time interval, the node reconstructs the original routing hash ID and injects the record in the routing process, which will deliver the record to the originally responsible node (in case problems of that temporarily unavailable node have been solved) or to another node for temporary storage. Once a temporarily stored record is re-injected in the routing process, it is deleted from the temporary storage.

6.2.2.5 Design Trade-offs

Several design trade-offs in those steps described above have been made, which impact the performance or scalability of SCRIPT. One major trade-off of the design is the use of a centralized element. A central element could reduce performance and could decrease the reliability of the solution. However, the decision to use a centralized element for some tasks was made due to the fact that using this approach a lower latency can be achieved compared to the same tasks being implemented fully in a distributed manner. The load on the CCR is expected to be small, since it is used only for management operations, such as identity provisioning, template management, or peer configuration. A node only contacts the CCR, when it is started (to receive an identity), when it receives an IPFIX record with an unknown template ID (to receive the template definition, its GTID, and routing function), and when it receives a new template definition (to map the newly observed template to an existing GTID).

Another design trade-off was concerned with the responsibility of the peer awareness task. In the implemented prototype the CCR periodically checks the node‘s availability and informs other nodes, when a node becomes unavailable. Designing peer awareness centrally allows for much faster reactions in case of a node being disconnected. A deployment of the solution presented here will not see more than several hundred nodes, thus, such a monitoring task can be performed easily by a single entity due to a reduced number of messages that the CCR has to process.

Finally, robustness of SCRIPT can be improved straight forward by introducing a secondary CCR, which mirrors the configuration and operation of the primary CCR and which can take over in case the primary CCR becomes unavailable.

6.2.3 Implementation

The SCRIPT distribution framework is implemented in C++ as a P2P overlay following the Kademlia protocol. Each SCRIPT node holds 64 buckets of other SCRIPT node addresses, for routing purposes. Each bucket can contain up to 20 addresses.

6.2.3.1 Application Support

Applications can be built on top of the SCRIPT framework by using an API provided by the framework. An application is started by registering it with the Controller Unit, by calling the method *registerApplication(templateId, application)*. The template ID passed in the registration call is used to identify those flow records, which will be passed to the application. If the application needs to receive more than one template, a separate registration call is required for each template ID.

An application needs to extend the class *LocalProcessor* and implement the methods *notify(char *msg)* and *process(sc_flowRecord *fr).* The notify method allows application-specific messages (such as configuration options, or queries) to be sent to an application during runtime. The process method is called, whenever a flow record with a template ID required by the application is received. A copy of a record is received using the process method.

6.2.3.2 Bootstrap Process

When starting a node, the IP address and port number of an existing SCRIPT node is needed for the bootstrap process. The first step is the retrieval of an identity from the CCR. The CCR maintains a list of already assigned identities and always tries to assign an identity in order to keep as much as possible a uniform distribution of assigned identities. The algorithm used by the CCR is to find the largest interval between two consecutive identities p and q, and choose the value [(p+q)/2] as the new identity.

During the bootstrap process a set of other existing SCRIPT nodes (IP addresses, port numbers, and node identities) is received from the bootstrap node and are used as an initial routing table. This information is exchanged using the Control Messaging component over UDP (User Datagram Protocol) messages. Whenever a node learns about another node in the network, two IPFIX sessions are created, one in each direction, between the IPFIX exporter of one node and the IPFIX collector of the other node, for exchanging IPFIX records. At the same time the new node is added to the appropriate k-bucket. These operations only take place when a new node connects to the network, so they only create a limited load.

6.2.3.3 IPFIX Collector

The IPFIX Collector component is implemented for UDP and SCTP (Stream Control Transmission Protocol) which are the two protocols favoured by IETF for IPFIX. The SCTP version, besides using the protocol preferred by IETF, allows better peer-awareness by using SCTP notifications when the status of an association between two nodes changes. When using SCTP, each time a node leaves SCRIPT all the nodes to which that node has an IPFIX session are immediately notified about the leave, so they can update their routing rules.

6.2.3.4 Routing

The routing function is used to send IPFIX records with similar characteristics to the same SCRIPT node. A routing function works as follows: L bits of the IPFIX record, starting at offset O are AND-ed with an L-bit mask V. The value resulting is passed through a hash function (the implemented prototype uses the BOB [61] hashing function) in order to receive the routing ID. A routing function is expressed as a set of its three elements: (L, V, O). Using such a combination of a mask and a hash function allows for a flexible manner to manipulate, which IPFIX records will map to the same routing hash ID.

After an IPFIX record is received by a node and its routing hash ID is calculated (or retrieved) a better candidate for processing that routing hash ID is searched in the P2P routing table. Once such a candidate node is found, the IPFIX record is appended to a queue of IPFIX records waiting to be sent to that node. A separate process continuously checks the queue status for each SCRIPT node and if the number of records reaches a predefined threshold an IPFIX packet with some of those records is sent to the respective SCRIPT node.

6.2.3.5 *Embedded Environment*

Cisco has recently introduced the Application Extension Platform (AXP) [65], which allows applications to run within a router. The implementation of SCRIPT has been compiled using the AXP Source Development Kit (SDK) and tested having two of these nodes running within two AXP cards. At the time of implementation and testing of the prototype in AXP, the SCTP protocol was not fully supported by these AXP cards yet, thus, the UDP alternative was applied as a transport protocol for IPFIX, when one or more nodes run on AXP cards.

6.2.4 Evaluation

In order to assess the applicability of the SCRIPT approach proposed a set of functional as well as performance analysis steps of the SCRIPT architecture and the implemented prototype was performed.

The main purpose of SCRIPT is, to recall its major benefit, to distribute IPFIX records to several machines according to rules required by an analysis application. This is achieved by organizing participating nodes in a P2P overlay and by using the P2P overlay information for distributing the IPFIX records. Using the API provided, applications can define routing functions according to their dedicated requirements. The same API allows for building analysis applications on top of SCRIPT and for receiving IPFIX records delivered by the SCRIPT framework.

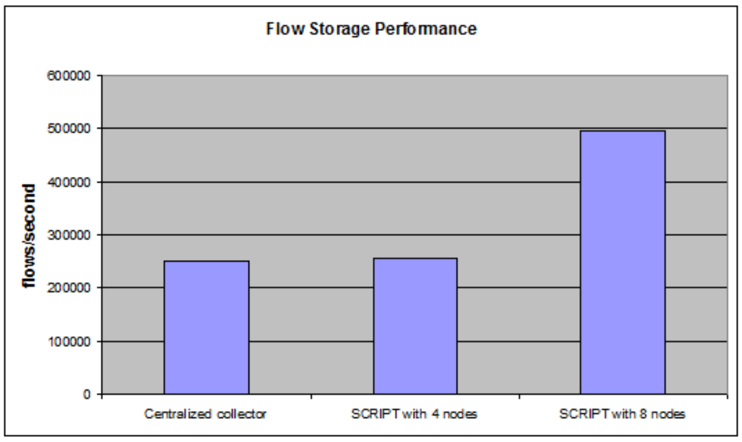


Figure 6-5. Flow Storage Performance.

Thus, the performance of the SCRIPT prototype is complex to be assessed, especially in comparison with other tools, since no such generic frameworks for distributed traffic data analysis exist. Therefore, the performance evaluation includes an evaluation of IPFIX records storage in a traditional, centralized collector, compared to the performance of a distributed collector built on top of SCRIPT. The tests were made using similar PCs with 3.6 GHz Intel processors, each having 4 GB memory. On the centralized collector the maximum rate of flow records that could be saved was 250,000 flows per second. Using SCRIPT running on 8 similar PCs in parallel a rate of 600,000 flows per second was achieved. In this evaluation, one stream of 150,000 flows per second was sent to 4 of the 8 nodes. Using only 4 nodes with SCRIPT the maximum flow rate that could be achieved in this prototype was 269,000 flows per second. These results are summarized in Figure 6-5.

During this evaluation it was observed that a single SCRIPT node can not process (in this case store in files) as many flow records as a similar centralized application running on the same node. The reason for this is that when running SCRIPT, some of the resources of a node are spent for calculating hash values and for the routing process, thus leaving less resources for the analysis application.

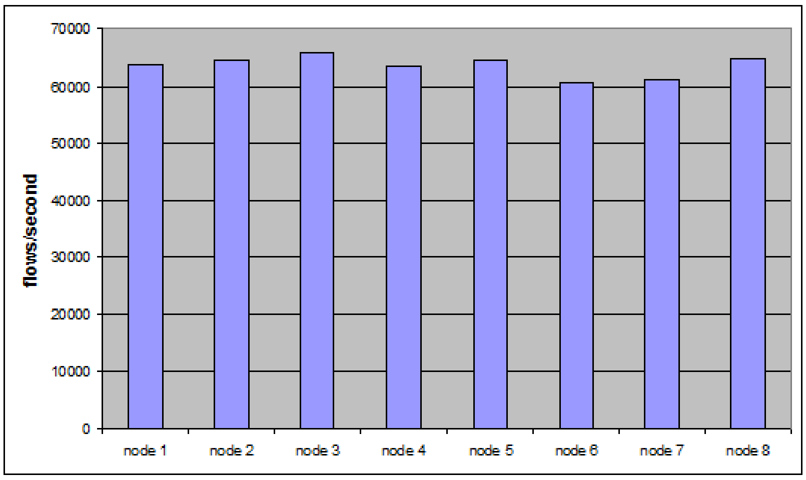


Figure 6-6. Record Distribution.

A second evaluation was performed to check, if SCRIPT distributes flow records equally between participating nodes. These results are shown in Figure 6-6. and outline that the average rate of flow records during a 60 seconds test using 8 SCRIPT nodes is at about 62,000. As it can be observed in Figure 6-6. , the maximum flow rate was 65,780 flows per second, while the minimum rate was at 60,535, resulting in a maximal deviation of 5% from the theoretical mean value.

6.3 Comparison between General and Specialized Information Sources When Handling Large Amounts of Network Data

Several techniques for monitoring network traffic are made available today, each one of them having a particular purpose and highlighting different aspects of network traffic information. Some of these techniques focus on collecting information about individual packets, such as tcpdump, while others focus on information about flows (*i.e.*, metadata information about sets of packets), such as NetFlow.

Independent of the level of abstraction involved (packets or flows), most of these techniques rely on storage points in order to store network traffic information. Network monitoring tools can use these storage points as information sources to retrieve network information. Information sources can be roughly divided into general purpose or specialized for a certain monitoring technique. MySQL is a well-known Database Management System (DBMS) and an example of a general purpose information source. It is extensively used by Web applications, but it has also been employed as an information source to store network traffic information. On the other hand, there are some specialized information sources available, such as NfDump, which stores network data into binary files without the context of a DBMS. NfDump is well known in the network community and commonly employed to collect network information.

Both information sources (MySQL and NfDump) deal relatively well with small amounts of data, but little is known when they have to handle large amounts. As an example, approximately 108GB of storage capacity, in a MySQL database, is required to store one week of network traces collected from SURFnet6, the Dutch national research network. In this context, the research question that motivates our investigation is: *What are the advantages and disadvantages of general purpose and specialized information sources for network monitoring when handling large amounts of data?* In order to answer this question, we performed a comparison between MySQL and NfDump, while observing response time when handling large amounts of data. For this comparison, we used 24 hours and 48 hours of network traffic from the University of Twente (UT) network, which was converted to both information sources storage formats.

6.3.1 Methodology

MySQL and NfDump were compared by measuring their query response times when handling large amounts of network data. While from NfDump's side, no optimization can be set up to speed up the execution of queries, MySQL provides the use of indexes, which results in more efficient access of data. In some cases, indexes can considerably increase the performance of a database table. The indexes are generally transparent to the end user, but MySQL offers the possibility to force or forbid the use of any of these. If an index does not exist or is not used, MySQL will perform a sequential scan over the whole table. However, if the index exists, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data. Nevertheless, if MySQL needs to access most of the rows, it is faster to read sequentially than to access the index, because this minimizes disk seeks.

Since indexes can affect the performance of a query, we tested MySQL both with and without indexes. This leads us to a total of four considered approaches:

1) **MySQL:** MySQL decides whether or not to use an available index.

2) **MySQL with indexes:** MySQL is forced to use a specified index.

3) **MySQL without indexes:** MySQL is forced to ignore any index.

4) **NfDump:** NfDump behaves as it is designed for.

In addition to these different approaches, we also took into account different ways for the considered information sources to access network data. Five case studies were considered in this case, each one of them presenting certain influence in the way the considered information sources treat the collected network data:

1) Listing: This case study causes the information sources to do a sequential scan over the network data, without doing any calculation or filtering. In fact, there is a filtering action on the start time attribute, but since we assume this as our basic situation (which we use in all queries), we consider it as part of the listing.

2) Listing + Filtering: The result of this case study is a subset of the result of the first case study. It contains only the flow records that have destination port 22. We chose to filter on this port because the traffic on port 22 is fairly distributed over the day and thus also distributed over the entire data set. The amount of traffic with destination port 80 for instance, would depend too much on the time of the day.

3) Grouping + Filtering 1: This case study groups all flow records of the "Listing + Filtering" case study by their source IP addresses and calculates the sum of octets for each group.

4) Grouping + Filtering 2: This case study is closely related to the previous one. Instead of grouping by just one attribute (source IP address), this query groups by five attributes. With this case study, we want to verify whether or not grouping by multiple attributes will require significantly more time than in the previous case study, with respect to our different approaches.

5) Listing + Filtering [Incremental Data]: All the case studies presented above are time-incremental. Despite the fact that network traffic with destination port 22 is commonly distributed over our data set, it is very likely that between 8 AM and 6 PM the traffic passing by the UT router is more than during the remainder of the day. We therefore formulated a last case study, which is data incremental instead of time-incremental. Consequently, we do not have to filter on start time anymore, but on the flow id attribute.

6.3.2 Evaluation

The results obtained in our analysis indicate NfDump as being the best solution to query large network data sets when observing response time. In all of our comparisons NfDump outperformed MySQL. NfDump offers several advantages compared to MySQL. First of all its query response times were much shorter than MySQL's response times in all of our measurements. We believe that this is because NfDump's input files are relatively small and can be concatenated to a larger data set. This is not possible with MySQL (unless table partitioning is considered). As a consequence, when a small part of a data set is needed for a query, MySQL has to browse through the whole data set. Since NfDump can do queries on smaller and individual data sets, its query response times are shorter. Another advantage of using NfDump, is the disk space needed to store the data; NfDump needs much less disk space than MySQL (with and without indexes, 22GB vs. 31 GB and 43 GB, respectively). One advantage offered by MySQL although, is its flexibility in accessing data. This is due to MySQL's wide variety of queries. This flexibility results in a richer set of statistics about network data.

6.3.3 Discussion

In spite of NfDump has outperformed MySQL in all of our comparisons, we observed that MySQL presented a constant response time after considering 7 hours of network data, whereas NfDump presented an increasing response time. To check if NfDump's response times would intersect MySQL's somewhere between 24 and 48 hours of network data, we enlarged our data set from 24 hours to 48 hours of data. The overall result was the same as with the smaller data set, in the sense that NfDump outperformed MySQL in all tests. However, while MySQL's query response times were doubled, NfDump's response times were slightly close to double. Taking this behavior as a pattern to even larger data sets, we can assume that NfDump's response times will always be shorter than MySQL's.

6.4 SNMP Usage: An Analysis of Additional Traces

The Simple Network Management Protocol (SNMP) is the de facto management standard for TCP/IP networks. It was firstly introduced in the late 1980s and currently is widely deployed. However, it is not entirely clear how SNMP is effectively used in different production networks. The network management community started to investigate the usage pattern of SNMP in 2007, and the first results were published in the same year. This investigation presents an analysis of a larger SNMP traces data set in comparison to the original work in order to determine the SNMP usage pattern. To the best of our knowledge, this represents an analysis of the most complete trace set for SNMP usage.

The traces used as input data for analysis were obtained from several locations. For this report we have analyzed 17 different traces from 9 different locations, including a private telecommunications company. In order to easily identify traces and locations, we have adopted the same naming scheme for traces described in [68]. This scheme contains two numbers, in which the first one describes the location and the second one the trace number. For example, l01t01 refers to trace 1 recorded at location 1.

Figure 6-7 provides a general characterization of the traces used in this research. It list number of messages in each traces and the distribution of the SNMP messages according to each version available. Percentages smaller than 0.0% are expressed as 0.0% and traces where none of the messages were of a particular version result in a dash in the respective column.

The table shows that all of the considered traces, except l13t01, are quite large in terms of the number of recorded SNMP messages. Another observation is that in the case of 10 out of these 17 traces 99.9% or more of the SNMP messages are of version SNMPv1. On the other hand, 7 traces have a majority of SNMPv2c messages. Finally, only one of these 17 traces had any SNMPv3 messages. However, since the number of SNMPv3 messages in l14t08 is so very small (just 3 messages), the percentage still remains far below 0.1%. These messages are probably the result of someone experimenting with SNMPv3. The distribution of the messages over the three versions is not significantly different compared to the distribution in [68], where there were also hardly any SNMPv3 messages found in the traces.

trace	messages	SNMPv1	SNMPv2c	SNMPv3
l01t01	71.501	100.0%	-	-
l01t03	11.332.845	99.9%	0.1%	-
l05t02	9.333.212	100.0%	-	-
l11t01	6.566.061	38.6%	61.4%	-
l13t01	6.455	16.5%	83.5%	-
l14t06	835.706	99.9%	0.1%	-
l14t07	3.215.572	100.0%	0.0%	-
l14t08	131.036.004	100.0%	0.0%	0.0%
l14t09	1.576.897	100.0%	0.0%	-
l14t10	1.037.907	100.0%	-	-
l15t02	1.844.512	15.8%	84.2%	-
l15t03	11.852.127	15.9%	84.1%	-
l15t04	8.980.759	25.9%	74.1%	-
l16t01	19.300.858	100.0%	0.0%	-
l16t02	306.598.103	100.0%	0.0%	-
l17t01	1.056.276	8.1%	91.9%	-
l18t01	30.487.406	19.2%	80.8%	-

Figure 6-7. Overview of the analyzed traces.

6.4.1 Discussion

The SNMP usage results were investigated after analyzing 17 real world traces. This work can be seen as an extension of [68], where the authors have analyzed a smaller number of traces. To do this analysis, we have used the same methodology and tools described in [68]. To the best of our knowledge, the achieved results present the most complete analysis of SNMP usage in real life production networks.

The reasons behind conducting this work include (i) to evaluate a larger trace set than in [68], (ii) to analyze newer traces than the previous work to verify if the results would change over time. The main conclusion is that there is no significant changes in the SNMP usage in our traces in comparison to the aforementioned work.

To achieve that, we have evaluated several aspects of SNMP usage. As in [68], we found that SNMPv3 is hardly used nowadays in real world networks. Of the 17 traces only 1 contained some SNMPv3 messages, despite the fact that the IETF declared SNMPv1 and SNMPv2 as historic and SNMPv3 as standard. Regarding PDU types, we also do not see significant differences compared to the results in [68]. The same can be stated about the most commonly referenced MIB: IF-MIB.

The main differences observed from our trace set are related to data type distribution and average response size. In our analysis, we found more occurrences of null types than in the original work. In addition, the average response message size we found is about 50 bytes, while in the original work was 300 bytes.

In this report we have also presented some extra analysis in comparison to the original paper. We have presented an analysis on varbind distributions, the impact of two different flow definitions and a more extensive look at traffic intensity in traces, but also in flows. As conclusions, we could observe that varbind of 1 is most common, which means that requests are generally very small. Secondly, the impact of the two definitions of a flow have been compared, which resulted in the finding that the introduction of the flow definition given in [63] seems correct (with respect to the made assumptions) and therefore does not result in a different separation of traces into flows compared to the previously, non-formalized definition of a flow.

6.5 Conclusions

IP flow records are frequently used in network management and traffic analysis, but classical flow collection and analysis architectures with centralized collectors have limitations regarding the scalability and performance in high-speed networks. The SCRIPT framework for IP traffic analysis introduced here addresses this problem by distributing flow records and analysis workload to multiple nodes. SCRIPT nodes build a Kademlia-based overlay to route and distribute flow records. If the overall load increases, new SCRIPT nodes can be added to the overlay on demand, requiring no manual configuration effort in the operation.

The SCRIPT framework also distributes the workload of analysis applications, since each SCRIPT node can run a part of the analysis task. Analysis applications, like delay measurement or asymmetric route detection, access the SCRIPT functionality over a well-defined API and the system can be extended with new applications. The SCRIPT framework uses a flexible routing function that can be specified according to demands of each analysis application separately. It builds on standard protocols and supports IPFIX and NetFlow-based data transfer that supports not only IP flow records but also per-packet information (e.g., packet header).

The SCRIPT framework has been implemented as a prototype and evaluated both on standard PC hardware as well as on Cisco AXP cards. The performance evaluations show that SCRIPT can increase the total number of flow records processed compared to a centralized solution and it scales with the total number of flow records exported in a network. The overhead introduced per SCRIPT node for flow record routing and relaying is low and the Central Configuration Repository (CCR) does not determine a bottleneck, since it is responsible only for management tasks, it does not participate in the flow record transfer, and SCRIPT nodes contact it rarely. As the evaluation reveals, the framework distributes flow records nearly equally among all nodes in the SCRIPT overlay, resulting in a fair balance of workload among all nodes.

Future work in this area shall focus on further performance evaluation both on standard PC hardware as well as on Cisco AXP cards. Additionally, analysis applications will be developed on top of SCRIPT and evaluated in terms of their performance.

In addition to the SCRIPT framework, this section presented a comparison between two techniques (NfDump and MySQL) for monitoring network traffic, and also presented some of the results of an SNMP trace analysis. For the first we have established that NfDump's response times are always be shorter than MySQL's. Regarding SNMP traces, we believe that, in addition to our analysis, further traces in different geographic locations should be taken into account. This evaluation should be also done periodically so the changes in the pattern usage can be observed.

7 P2P Collaboration and Storage (P2P COST2)

7.1 Introduction

The popularity of devices with IP connectivity, both stationary and mobile, combined with growing reachability of such networks, increases the demand for high-quality video transmission over the Internet [87]. In order to deliver high Quality of Experience (QoE) to the end user, video streaming requires very large bandwidth at the server side, and is very sensitive to congestion, especially in a live transmission. To alleviate those problems, peer-to-peer (P2P) technology has been successfully applied to video streaming, reducing bandwidth and management cost for the distributor and adding scalability as the number of users watching the stream increases.

LiveShift [81] is a P2P application that combines live video streaming and video on demand (VoD). While a P2P approach is used to distribute the stream among peers, the information is selectively recorded by the peers, enabling users to watch recorded content from other peers. It exploits often underutilized resources, such as disk space and network bandwidth of existing desktop computers, harnessing a potentially large and self-scaling system. VoD in LiveShift consists on three phases: (1) recording live streams in a coordinated manner yet with low overhead, (2) finding peers responsible for the recording of the requested time slots, and (3) retrieving the video stream from the responsible peers in an optimal manner.

For delay sensitive P2P video streaming, there is a natural distribution hierarchy induced by the increasing delay from the stream originator (the *peercaster*). Thus, if one considers the *delay space* on which the P2P system is embedded, algorithms that provide distribution topologies that “spread out" from the peercaster will be more efficient than those for which connections are made in all directions of the delay space. Furthermore, such a directed swarming protocol must reduce stream lag while at the same time rewarding peers that contribute more to the network and punishing free riders. Current proposed incentive mechanisms are either susceptible to many kinds of Sybil attacks, fail to distinguish between the incentives of the peercaster and consumer peers, or do not consider lag minimization explicitly.

This section describes a protocol for Sybil-aware directed swarming for the distribution of live content. This mechanism organizes peers to reduce delay and increase the number of stream recipients without a central organizer, even in the presence of churn and unreliable users. The system is resistant to certain classes of known mechanisms for "lying peers" and Sybil attacks. Peers are rewarded for their contributions to uploading the stream by moving closer to the peercaster, giving them both increased liveness and stability. Although Sybil attacks are still possible, they produce a very limited benefit for the attacker.

7.2 Related Work

Early P2P systems, such as Gnutella and Napster, did not implement any incentive mechanism. They, consequently, suffered from the dissemination of *free-riders*, which are peers that act selfishly, consuming resources without contributing to the system [73]. Such behaviour leads to a widespread degradation of video stream quality.

Tit-for-tat (TFT) [75] is a game-theoretic strategy used to reach Pareto efficiency in the prisoner's dilemma. In the classic TFT scheme, a peer is only able to consume as much resources as it provides to another peer. Each peer in a TFT scheme must keep a *transaction history* to track past resource exchanges with other peers.

BitTorrent [79] is a popular P2P file-sharing application which implements a TFT-like incentive mechanism. A strict piece-for-piece TFT was discovered to hinder performance of the system, since the resource to be optimized is upload bandwidth. The success of TFT in BitTorrent led to it being used also in some P2P video streaming systems, *e.g.* BiToS [86]. TFT, however, is shown not to perform well in P2P video streaming due to the additional delay and bandwidth constraints imposed by it [74][78][85]. TFT requires symmetry of interest between peers, not being applicable to the peercaster, which has no way of determining which peers should be forwarded the stream first-hand.

Transitive TFT mechanisms [76][80][88] can effectively reduce the risk of malicious attacks such as free-riding, collusion and Sybil attacks. They work by allowing peers to share their past resource exchanges with other peers. However, they are also not applicable to the peercaster, since it does not receive streams from other peers. In addition, these techniques suffer from convergence problems stemming from their limited view of the contribution state of the system, as well as their use of potentially outdated information.

Give-to-Get (G2G) [84] is an incentive mechanism built for P2P VoD. Peers rely on reports by other peers to decide which is contributing more to the network and, therefore can penalizing free-riders in favor of well-behaving peers. The main problem it does not take into account that peers may collude or be Sybils.

Related work is summarized and compared in Table 7-1.

Table 7-1. Related work comparison.

Incentive Mechanism	***Collusion & Sybil-proof***	***Applicable to Peercaster***	***Application***
TFT	Yes	No	BiToS
GTG	No	Yes	Tribler
TTFT	Yes	No	*none*

7.3 Concepts and Requirements

This Section presents background information about LiveShift and P2P video streaming. It also presents the requirements and assumptions that we require from our proposed approach.

7.3.1 LiveShift Concepts

LiveShift is based on concepts presented in [81], but with modifications with regard to stream distribution and storage policy. Stream distribution is done in a mesh-pull fashion, in order to fully utilize upstream bandwidth of each peer and cope with churn.

LiveShift, differently to popular applications such as [71], implements a server-free architecture. The only single point of failure is the peercaster - the peer that is the original stream source. Channel and tracker information are distributed in a Kademlia-based [83] DHT [72]. The deployment of superpeers to overcome limitations caused by asymmetric bandwidth is possible.

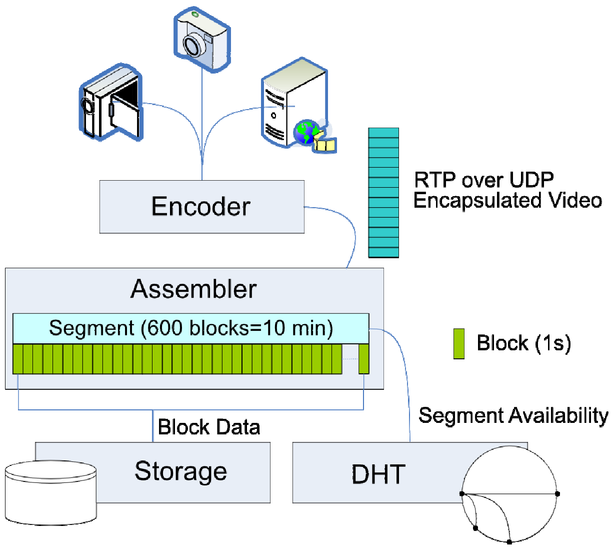


Figure 7-1. Blocks and segments in LiveShift.

In mesh-pull streaming, the source peer divides the stream in chunks, which are exchanged among peers in the P2P network. To address the trade-off in chunk size - a larger piece causes less overhead, but more delay - LiveShift implements a dual approach, as pictured in Figure 7-1. A video stream is divided in small pieces (of 500 milliseconds of length) called *blocks*, which are exchanged among peers. A group of 600 pieces forms a *segment*, which totals to 5 minutes of video. Segments are are identified by unique Segment Identifiers, which are announced on a DHT by peers that possess at least one block in the segment. The system borrows several concepts from BitTorrent, since it has shown to successfully take into account many important real-life limitations, *e.g.* with regard to number of open TCP connections. A simplified description of the main procedures follows; full details are presented in 7.4.

Peer and content discovery are done through the DHT. Peers exchange *block maps* showing which blocks they can offer in a segment. Peers request upload slots from peers that have interesting blocks. Received requests for upload slots are put in a queue until they are granted. Peers may only send block request messages to peers while they have been granted upload slots. The Block Scheduler is responsible for selecting the blocks a peer must download and also for selecting which peers to download each block from. After locating which peers have the desired blocks, the content can be downloaded by requesting each individual block.

7.3.2 Requirements

We propose the following engineering objectives for our directed swarming mechanism:

Fostering cooperation

Free-riders and peers that contribute little to other peers must have less chance of obtaining video blocks from other peers on time, therefore obtaining lower quality of experience.

Resistance to protocol deviation

Peers that do not follow the protocol correctly , *e.g.* by reporting false information or creating Sybils, must be punished by having less chance of obtaining video blocks form other peers.

Decentralized operation

The solution must be fully decentralized in order to be compatible with the overall architecture.

Robustness

Since P2P systems are heavily affected by churn, the system must cope with the fact that peers will join and leave the system unexpectedly.

Scalability

The solution must scale with regard to the number of peers in a swarm.

Low overhead

Video streaming is very bandwidth-consuming and sensitive to delay, therefore network overhead introduced by the incentive mechanism must be kept to a minimum.

7.3.3 Assumptions

The following assumptions are present in the design of the protocol:

1. Peers who wish to watch a stream desire to do so with as low delay as possible. They desire to move “closer" to the peercaster to be more reliable and more “live".
2. The peercaster is motivated to get as many “genuine" viewers as possible for its content, in order to efficiently distribute the stream to as many peers as possible. Other peers in turn, should recursively be able to transmit the stream to as many peers as possible.
3. Peers may try to cheat by lying about their contribution or creating Sybils that send optimistic reports about their contribution.
4. It is hard for a peer to receive and respond to messages sent to widely different parts of the IP space.

7.4 Protocol definition

The proposed directed swarming mechanism operates by rewarding peers with lower delay streams if they provide upload resources to the overlay by uploading the stream to as many “genuine" viewers as possible. To this end, each peer is given a particular *score*, which is calculated by taking into consideration the recursive contribution of each peer (the whole series of downstream uploads that is enabled by their initial uploading). Thus, this *score* represents the effectiveness of the peer at passing on the stream. By construction, scores will be calculated on the basis of the scores of those peers to which a peer uploads. Any peer will connect preferentially to those peers with highest score, since doing so will increase their own score. Furthermore, a peer will gain access to higher levels of service only by having a high score itself. Consequently, the swarming decisions of peers are expressed through two different kinds of decisions: peers attempt to connect to lower delay peers and move closer to the peercaster, while accepting or rejecting connection attempts according to the *score* of the connecting peer.

The main objective of this swarming algorithm is to build an overlay in which the peers with the highest overall contribution are placed near the peercaster. This will act as an incentive to provide resources to the overlay, while at the same time helping reduce the stream lag.

In short, the protocol involves the following:

1. Each peer is given a score, which takes into consideration the recursive contribution of each peer to the entire P2P network in a time period
2. In order to increase their own score, peers need to upload to peers with high score
3. Peers desire to increase their own score in order to move “closer" to the peercaster, be more reliable, and experience shorter delay
4. The peercaster desires to efficiently distribute the stream to as many peers as possible, therefore it chooses high-scoring peers to upload their stream to
5. A Sybil-resistant system for scoring which rates peers according to their success at uploading (a higher score improves the chance of getting close to the peercaster).
6. A distributed auditing system which checks peers claims for their score are reasonable and penalizes those peers who “cheat".

Table 7-2 shows important quantities that are used in the system.

Table 7-2. Important quantities used in the system.

$d_i(t)$	application-layer average hop count between the peercaster and peer *i* at time period *t*
C_i	set of candidate peers, which have blocks in a segment in which peer *i* is interested
N_i	set of peers in which a peer *i* is waiting in the upload queue
D_i	set of peers a peer *i* is downloading from, that is, set of peers which are granting *i* an upload slot
$\overline{D}_i$	the maximum size if D_i
$D_i(t)$	set of peers a peer *i* has downloaded from in time period t
Q_i	set of peers that are in an upload queue, waiting to get an upload slot from peer *i*
$\overline{Q}_i$	maximum size of Q_i
U_i	set of peers a peer *i* is uploading to
$\overline{U}_i$	the maximum size of U_i, that is, the number of upload slots

$U_i(t)$	set of peers a peer *i* has uploaded to in time period *t*
$S_i(t)$	the score of peer *i* in time period *t*
$d_i(t)$	application-layer average hop count between the peercaster and peer *i* at time period *t*

Tweakable variables used in the system include:

- T_a – a time period for audits and score updates. (must be globally known)
- T_d – a time period downloaders use to reconsider who to request from. (can be different for each peer)
- T_u – a time period uploaders use to reconsider which requests to honour. (can be different for each peer)

7.4.1 Requesting and Granting Upload Slots

Peers download video blocks from several other peers. Each peer *i* uploads to a set U_i of peers at a time – this set grows to user defined number $\overline{U}_i$. Each peer *i* tracks, for each time period $t \in T_a$, its own score $S_i(t)$, which is influenced by the scores reported by the peers it has uploaded to during T_a, $U_i(t)$. Score calculation is presented in Section 7.4.3. Initially, peers are assumed to behave according to the protocol and send truthful reports; this assumption is later relaxed.

When a requester peer *r* tunes into a channel, it initially looks for any other peer willing to send the stream to it. In order to do so, it looks up in the DHT for a specific key, which is created based on the unique Segment Identifier that has the blocks which need to be downloaded first. The DHT lookup results in a list of candidate peers $c_{0..n} \in C$ that are offering blocks in the given segment. Peer *r* sends US_REQUEST (US stands for upload slot) messages to a random subset of *C*. The message includes the score of the requesting peer S_r, which is initially zero.

When a peer $s \in C$ receives a US_REQUEST message from a peer *r*, it attempts to place peer *r* in its upload queue Q_s. If $|Q_s| < \overline{Q}_s$, the upload queue is not full yet, so peer *r* is sent a US_WAIT message, with the Block Map for the requested segment, S_s, and a timeout value T_u. If the upload queue is full, peer *s* may pre-empt a peer *p* if $\exists p \in Q_s : S_p < S_r$. In this situation, peer *p* is sent a US_REJECT message, while peer *r* is sent a US_WAIT message. Finally, if there is no peer to be pre-empted, peer *r* is sent a US_REJECT message.

US_WAIT messages contain a timeout value; if after this time no US_ACCEPT or US_REJECT is received then the request times out and a US_REJECT is assumed.

Peer *s* has a fixed number of upload slots $\overline{U}_s$, each of which constantly fetches a peer $h \in Q_s : \nexists i \in Q_s : S_i > S_h$ from the queue and grants it an upload slot. Peer *h* is sent a US_GRANTED message, which contains a timeout value that indicates how long the slot will be granted for. The timeout value is to be used as a reference only, since *s* may send additional US_GRANTED messages to extend the granted time, or a US_WAIT message, indicating the slot has been revoked and the peer *r* is back in the queue, waiting for a slot.

Every time peer *s* obtains a new block, it sends a HAVE message to all peers in $Q_s \cup U_s$, for them to update their block maps accordingly. This way, they are able to keep an up-to-date view on the system.

When peer *r* receives a US_WAIT message from peer *s*, it places *s* in the set N_r, together with $d_r(t)$ and the timeout value.

When peer *r* is granted an upload slot from peer *s*, it will be allowed to send BLOCK_REQUEST messages to *s* in order to receive the video block in BLOCK_REPLY messages.

Once *r* has at least one upload slot, it will attempt to improve its position in the overlay by decreasing its application-level hop count with regard to the peercaster $d_r(t)$. Every time period T_d the peer attempts to get an upload slot from a new candidate peer $n \in C_r$ with lower $d_n(t)$ than the ones currently granting one. An attempt to move much closer to the peercaster is unlikely to succeed.

If $d_r(t) < d_n(t)$, *n* does not represent any improvement, another peer is sought and a US_REJECT message is sent to *n*, which will remove *r* from its upload queue. Otherwise, if *n* replies with US_WAIT, *n* is added to N_r.

If a peer finds itself with no upload slots, it will switch immediately to the initial protocol from the first paragraph. Instead of choosing from candidate peers peers randomly, however, the peer will have a list of previous uploaders which could be tried (in order of their application-level hop count).

7.4.2 Requesting Video Blocks

The procedure for requesting video blocks is relatively simple. LiveShift performs block selection first, then peer selection for every selected block. Every 500 ms, a peer selects the next 10 video blocks, counting from the current playing time, which have not already been downloaded, and which have not been requested in the last 1000ms.

For each selected block, a peer *r* selects a random peer from D_i which has the block. Peer *r* knows which blocks they have, since it receives from them a block map in the US_WAIT message, and, after that, updates via HAVE messages. Peer *r* may also perform a delay-based peer selection instead of a random one, by measuring how long each peer in D_r takes to reply with a block and keeping a ranking.

A peer *s* receiving a BLOCK_REQUEST message from peer *r* will only reply positively with a BLOCK_REPLY message if peer $r \in U_s$ and if it currently has the block – otherwise, the message will be silently discarded.

7.4.3 Score

The score of each peer represents its contribution to the P2P network as a whole. Therefore, it takes into account the contribution (upload) of the peer itself and, recursively, all the peers which are benefiting from it.

7.4.3.1 Basic score

First, a basic score system is defined. The concept is later extended to take the possibility of Sybil attacks into account. In a general sense, each peer has a score $S_i(T)$, which is a measure of its contribution in a given time period T_a.

Desirable qualities for the score include:

- $S_i(t)=0$ if peer i uploads to nobody in period t.
- $S_i(t)=N$ if peer i uploads either directly or indirectly the entire stream to N other peers, which do not further upload the stream.
- $S_i(t)\geq S_j(t)+1$ if peer i uploads the entire stream to peer j in period t.

Let $0\leq p_{rs}(t)\leq 1$ be the proportion of the stream which peer s uploaded to peer r in period t. For simplicity we will drop the subscript. The score should be a measure of the proportion of the stream the user has contributed including indirect contribution to all peers it has uploaded to – that is, if a peer s passes uploads a complete stream to one peer who uploads to one further peer and so on for M peers, then the original peer s is credited for all M uploads. This would give an incentive for peers to choose wisely the peers they grant upload slots to.

Initially, for simplicity, we consider the situation where each peer downloads the complete stream $\sum_{j\in D_i} p_{ij} = 1$for all i. Now, imagine the circulating system where each peer (including the peercaster) uploads a proportion $1/N+1$ to the peercaster itself. This could be viewed as an ergodic discrete time Markov chain with $N+1$ peers numbered 0 to N. The peercaster (peer 0) has transitions P_{0i} to all i including itself with transition probability $1/(N+1)$. The transition $P_{ij}=p_{ji}$. Now the equilibrium probabilities π_k for a state k are given by the balance equation:

$$\pi_i = \pi_0/(N+1) + \sum_{c\in D_i} \pi_c p_{ci}, \tag{7.1}$$

where the first term from the peercaster and the subsequent terms are peers from $D_i(T)$. This π_i can be thought of as the proportion of times a random video block flows through peer i on its loop around the system.

Now we wish to convert π_i to some normalized score with the property that the score can be simply totalled such that the score of the peercaster is N (since the peercaster is responsible for uploading the entirety of the stream to N peers) and the score is linearly related to π_i. If we use $S_i=(N+1)\pi_c/\pi_0-1$ (or $\pi_c=(S_c+1)\pi_0/(N+1)$ then (7.1) becomes:

$$(S_i+1)\pi_0/(N+1) = \pi_0/(N+1) + \sum_{c\in D_i} (S_c+1)\pi_0 p_{ci}/(N+1) \tag{7.2}$$

which simplifies to:

$$S_i = \sum_{c \in D_i} (S_c + 1) p_{ci}. \tag{7.3}$$

This score can be though of as the score for node *i* is the sum over all the peers *c* which downloaded from node *i* of the score for that node plus one for the node *c* itself multiplied by the proportion of the stream sent to that peer. An obvious case is that S_i=0 if D_i=∅ that is a peer which uploads to no others has a score of zero.

7.4.3.2 Score with IP binning

The basic score above is modified slightly to reduce the possibility of a Sybil attack. In particular we consider here that it is difficult for a peer to produce sybils which pass audit (see Section 7.4.5) at different places in the IP address space. However, it may be easy for a peer to produce a number of sybils with the same (or similar) IP addresses – either by pretending to be many different peers sharing an IP (behind a NAT) or by controlling a small block of the IP space. The concept behind the modified score is to divide the IP space into bins and to track contribution to each. The objective is to reduce the return obtained by a peer which creates Sybils.

For each time period $t \in T_a$ each peer keeps track of contributions to each section of the IP space using the IP binning vector $B(T) = (i_1(T), i_2(T), \ldots, i_m(T))$ where $m \in N$ is an integer chosen to be an appropriate number of bins for the IP space. One obvious scheme, for example, would be to have $m=2^8=256$ and have one bin for each possible first octet of an IPv4 address.

A peer *k* tracks the uploaded contributions reported by $U_k(t)$. So, dropping the *T* and taking B_k to be the IP bin information for peer *k* then the equation (7.3) becomes:

$$B_k = \sum_{j \in N_k} (B_j + C(j)) p_{ij}, \tag{7.4}$$

where *C*(*j*) is an IP binning vector which is zero except for a one in the appropriate bin for peer *j* in the IP space. Similarly to the raw score, a peer *k* which uploads to nobody has $B_k = (0, 0, \ldots, 0)$.

The aim now is to construct a score which considers the IP binning vector and reduces the score if the vector is unbalanced. That is, the score is modified downwards if the peer is claiming too many other peers in a given section of the binned IP space (possible Sybils). The modified score is calculated from the IP binning vector *B* using the following formula:

$$S = \sum_{j=1}^{m} \begin{cases} i_j & i_j \leq T(B) + 1 \\ \alpha(B)(2 - \exp(T(B) + 1 - i_j) & i_j > T(B) + 1, \end{cases} \tag{7.5}$$

where *T*(*B*) is the point at which the number of peers in a given bin in the binned IP space is considered suspicious. It is given by *T*(*B*) is the maximum of the smallest α*m* bins in *B* and α∈(0,1) is a parameter, 0.95 would seem reasonable as a start.

7.4.3.3 Improved IP binning

The equal-sized bins produced as described above would have the problem that, since the IP space is not uniformly assigned, many bins would naturally be always empty, while some other bins may concentrate many IP addresses. In order to address this issue, the currently assigned IPv4 addresses were obtained. Figure 7-2 plots the IPv4 assignment in 65536 bins. Currently, only 21.61% of the IPv4 space is assigned – the rest are either not yet assigned or reserved.

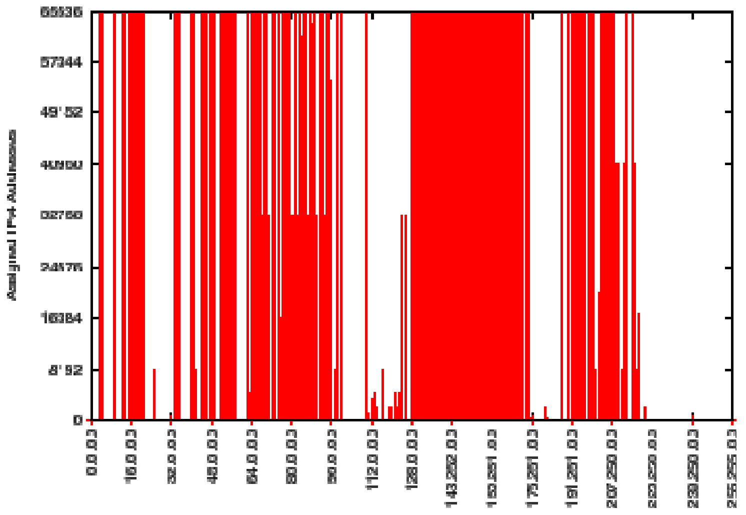


Figure 7-2. Currently Assigned IPv4 Addresses.

IP binning is modified as follows. Let the vector $K=(k_1,k_2,\ldots,k_m),0\leq k_i\leq 1$ represent the share of assigned IP addresses in each bin. *B* is then modified to use those coefficients into account. The new vector $I=B\times K$ is used instead of *B* in equation (7.5).

With m=65536 and the current IPv4 assignment information, only 15195 elements of K would have value greater than 0. Peers with IPs in those bins would have no score, since their k_i coefficient would be 0. Those peers either have invalid IP addresses, or the list is incorrect, possibly outdated.

Although this mechanism is explained in IPv4 terms, it is possible to use it also with IPv6 addresses, since the assumption that it is expensive to control several different subnets. The main difference would be that a larger number of bins would be necessary, but the details of implementation in IPv6 are future work.

7.4.4 Score update

Peers calculate their own scores based on the scores of peers they have uploaded to in a given time period. Scores are updated at an interval $T_s<T$. At intervals T_s a peer will send a SCORE_UPDATE message with its score for an interval to all peers in $D_i(T)$ according to the following rules.

For each time period *T* peers must track for each peer it downloaded from ($D_i(T)$), the proportion of the stream it downloaded, which cannot sum to more than one, and for each peer in $U_i(T)$ the proportion of the stream in that time period it uploaded to each peer. They must also track the reported IP bin vectors from each peer which downloaded from it in each time period.

At the end of every period T_s, a peer will send a SCORE _UPDATE message upstream to peers in $D_i(T)$ in the following circumstances:

- If the peer has actually downloaded content in a given T period from a peer it will send an updated score for that period to the peer it downloaded from.
- If the peer has received a score update from a child which caused it to change its score in a given period from the last reported score by more than β percent (β is a parameter).

A SCORE_UPDATE received from a peer in $U_i(t)$ may cause a peer to pass a revised score upstream at the end of the next T_s according to the same conditions. In time the scores should converge to the scores in the previous sections.

7.4.5 Auditing behaviour

Peers could cheat by falsely declaring their score. Therefore, each peer *a* which has $|U_a|>0$ performs a lightweight audit using the following protocol.

A peer *a* will audit another peer *u* immediately the first time it receives a US_REQUEST message from it that would make it into the upload queue. In addition, every time period T_a the peer audits some peers $u\in U_a(T_a)$. Let the score claimed by each $u\in U_a(T_a)$ be $S_{u1}, S_{u2}, \ldots, S_{u|U_a|}$. Peer *a* audits *u* with probability $S_u / \sum_{j=0}^{|U_a|} S_j$.

To audit a given peer *u*, peer *a* sends an AUDIT_REQUEST message to peer *u* to send an AUDIT_REPLY message with its $U_u(T)$, that is, the IDs of the peers it has uploaded to, along with the share of the stream sent to each peer and a single byte from the stream at a recent time *t*. If this information is returned correctly and if the score S_u matches the score reported by peers in U_u, then the peer has “passed audit" and the audit continues for each peer in U_u until it reaches peers claiming no uploads.

If a peer *f* fails audit then the auditing peer *a* reports the fact to the known peers which are uploading to it (D_f inferred) via an AUDIT_FAILURE message. *f* will be pre-empted from their upload slots unless the parent has spare bandwidth. Its score will be assumed to be zero until it passes a subsequent audit. In addition, the peer which detected the failure will apply a penalty to the estimated score all peers on the upload path to this failed peer, since they have not properly performed audit. This creates an incentive for these nodes to perform their own audits – if they do not detect a fraud they are likely to be penalized themselves.

When there is a mismatch of information between two peers, it is important to tell which peer is lying. Let us imagine that there is an audit trail where a peer O (for origin) has audited peers $A\rightarrow B\rightarrow C\rightarrow D$ and *D* fails. We have a "punishment factor" *R* (for retribution if you like) > 1. *A*, *B* and *C* are informed of the failure and privately revise their opinion of their child node. So let us assume that *D* has claimed a score S_D but *D* failed audit. *C* revises their private opinion of *D*’s score to zero and *D* may be dropped by *C* if other nodes apply. *B* revises their private opinion of *C*’s score by removing Rp_CS_D where p_C is the proportion of the stream *C* uploaded to *D* in the period. This may cause *B* to drop *C*. A revises their private opinion of *B*'s score downwards by $Rp_Bp_CS_D$ and *O* revises their private opinion of *A*'s score by $Rp_Ap_Bp_CS_D$. The damage done by failing an audit incentivizes peers to perform audits and find cheaters (before someone upstream from them does). The score revision is only made privately by the people on the audit trail BUT these are people who can directly affect the cheater.

7.5 Conclusions

The proposed incentive scheme is yet to be evaluated. The objective of the evaluation is to prove the requirements stated in Section 7.3.2.

Foster cooperation

It must be shown that free-riders experience low QoE, while powerful peers which contribute significantly have great QoE.

Prevent protocol break

Peers which lie about their contribution or the contribution of other peers get caught fast enough by the auditing mechanism. Creation of Sybils does not pay off.

Decentralization

This is by design – the mechanism is fully-distributed. Not having a central entity avoids the creation of a central point of failure and improves scalability.

Robustness

The system must be shown to work with reasonable levels of churn.

Scalability

The solution must scale with regard to the number of peers.

Minimize overhead

Tradeoffs in frequency of score update and auditing must be clearly shown.

8 Summary and Conclusions

This document describes the collaborative work carried out by EMANICS WP9 partners during the last phase of the project. The presented work addresses various key areas of autonomic management and provides state of the art technologies and solutions to many of the challenges surrounding this topic.

Our work on service management demonstrates how peer-to-peer technologies and artificial intelligence techniques can be used to automate fault recovery operations in today's complex and highly inter-dependent IT operation environments. The two main aspects of the proposed approach are the advanced reasoning method for the automated discovery of the distributed fault recovery knowledge, and the planning algorithm and corresponding information models for the automated composition of fault recovery plans. A policy-driven network reconfiguration methodology for automating the recovery from link failures is also proposed.

We envisage that federation mechanisms will play an important role in future networking scenarios where resources will be shared between multiple autonomous entities. To this end, we have developed a methodology for the orchestration of autonomous entities as well automated resource negotiation models. These aim at coordinating the operation of two or more autonomic control loops, and ensure effective sharing of resources among competing entities.

Policy-based techniques are intimately linked to the principles of autonomic management, since they can provide the means for dynamic system adaptation in response to emerging conditions. We have identified the main policy types that are essential for realizing the functionality of an autonomic element, and classified them in a policy continuum with five layers of granularity. Specific policies to cope with the main characteristics of autonomic elements have been proposed.

Work in WP9 has also addressed the management of limited capability networked devices. The proposed self-healing and protection mechanisms aim at detecting and handling potential faults in wireless sensor networks by means of masking or isolation. Furthermore, a data sharing scheme based on a publish/subscribe architecture has been proposed, which can result in efficient memory usage and in extended life-time of sensor nodes.

Distributed and autonomic real-time traffic analysis has proposed a novel framework for IP traffic analysis that overcomes the scalability and performance limitations of centralized approaches in high-speed networks. The SCRIPT framework distributes flow records and analysis workload to multiple nodes, based on a peer-to-peer overlay for routing and distribution. The framework has been implemented as a prototype and evaluated both on standard PC hardware as well as on Cisco AXP cards.

Finally, the peer-to-peer technology has also been considered as an alternative to centralized approaches in the context of video streaming over the Internet. Research focused on a protocol for Sybil-aware directed swarming for the distribution of live content. The approach overcomes the problems of currently proposed incentive mechanisms and is resistant to certain classes of known mechanisms for lying peers and Sybil attacks.

9 References

[1] P. H. Winston, “Learning and Reasoning by Analogy,! Communications of the ACM, 23(12):689–703, 1980.

[2] P. K. Koton, “Using Experience in Learning and Problem Solving,” PhD thesis, Laboratory of Computer Science, Massachusetts Institute of Technology, 1988.

[3] J. G. Carbonell, “Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition,” In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, Machine Learning: An Artificial Intelligence Approach. Morgan Kaufman Publishers, California, USA, 1986.

[4] M. M. Veloso and J. G. Carbonell, “Derivational Analogy in Prodigy: Automating Case Acquisition, Storage, and Utilization,” Machine Learning, 10:249–278, 1993.

[5] P. Cunningham, D. Finn, and S. Slattery, “Knowledge Engineering Requirements in Derivational Analogy,” In Proc. 1st European Workshop on Topics in Case-Based Reasoning, pages 234–245, London, UK, 1994. Springer-Verlag.

[6] B. Blumenthal and B. W. Porter, “Analysis and Empirical Studies of Derivational Analogy,” AI Journal, 67(2):287–328, 1994.

[7] R. Bareiss, “Exemplar Based Knowledge Acquisition: A Unified Approach to Concept Representation, Classification, and Learning,” Academic Press Professional, Inc., California, USA, 1989.

[8] H. Tirri, P. Kontkanen, and P. Myllymäki, “A Bayesian Framework for Case-Based Reasoning,” In Proc. 3rd European Workshop on Advances in Case-Based Reasoning, pages 413–427, London, UK, 1996. Springer-Verlag.

[9] A. F. Rodriguez, S. Vadera, and L. E. Sucar, “A Probabilistic Model for Case-Based Reasoning,” In Proc. 2nd International Conference on Case-Based Reasoning Research and Development, pages 623–632, London, UK, 1997. Springer-Verlag.

[10] P. Gomes, “Software Design Retrieval Using Bayesian Networks and WordNet,” In Proc. 7th European Conference on Advances in Case-Based Reasoning, pages 184–197. Springer-Verlag, 2004.

[11] D. Heckerman, J. S. Breese, and K. Rommelse, ”Troubleshooting Under Uncertainty,” Technical Report MSR-TR-94-07, Microsoft Research, January 1994.

[12] E. Lazkano and B. Sierra, “BAYES-NEAREST: A New Hybrid Classifier Combining Bayesian Network and Distance Based Algorithms,” In Proc. 11th Portuguese Conference on Artificial Intelligence, pages 171–183, Berlin, Germany, 2003. Springer-Verlag.

[13] J. Pearl, “Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference,” Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[14] H. M. Tran and J. Schönwälder, “Fault Representation in Case-Based Reasoning,” In Proc. 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, pages 50–61. Springer-Verlag, 2007.

[15] P. Szolovits and S. G. Pauker, “Categorical and Probabilistic Reasoning in Medical Diagnosis,” Artificial Intelligence, 11(1-2):115–144, 1978.

[16] T. Cover and P. Hart, “Nearest Neighbor Pattern Classification,” IEEE Transactions on Information Theory, 13(1):21–27, 1967.

[17] H. M. Tran, G. Chulkov, and J. Schönwälder, “Crawling Bug Tracker for Semantic Bug Search,” In Proc. 19th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, pages 55–66. Springer-Verlag, 2008.

[18] Networking Forum: http://www.computing.net/networking/wwwboard/wwwboard.html. Last access in May 2008.

[19] M. Ghallab, D. Nau, P. Traverso, “Automated Planning – Theory and Practice,” Morgan Kaufmann, 2004, ISBN: 1-55860-856-7

[20] Kutluhan Erol, James Hendler, and Dana S. Nau, “UMCP: A sound and complete procedure for hierarchical task-network planning,” In Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS 94), pages 249–254, 1994.

[21] S. Blake et al., “An architecture for Differentiated Service,” RFC 2475, Dec. 1998.

[22] P. Psenak et al., “Multi-Topology (MT) Routing in OSPF,” IETF RFC 4915, June 2007.

[23] D.C. Verma, “Policy-Based Networking,” New Riders Publishing, ISBN 1-57870-226-7, pp. 155-165, 2000.

[24] D. Nau et. Al., Applications of SHOP and SHOP2, IEEE Intelligent Systems, 31-41, 2005

[25] Boudjemil, Z. (editor), Autonomic Internet Project, Deliverable D3.1 – AutoI Information Model, 2009.

[26] Galis, A. and Mamatas, L. (editors), Autonomic Internet Project, Deliverable D4.1 – Initial Management Plane Design, 2009.

[27] Cheniour, A. and Lefevre L. (editors), Autonomic Internet Project, Deliverable D5.1 – Initial Service Enablers Plane Design, 2009.

[28] Pujolle, G. (editor), Autonomic Internet Project, Deliverable D2.1 – Initial OP Design, 2009.

[29] EU-IST Autonomic Internet (AutoI) Project – http://ist-autoi.eu/autoi.

[30] Future Internet Design (FIND) Program – http://www.nets-find.net/.

[31] Global Environment for Network Innovation (GENI) Program - http://www.geni.net/.

[32] S. Kraus, “Automated negotiation and decision making in multiagent environments,” Lecture Notes in Artificial Intelligence 2086, pp. 150-172, 2001.

[33] H. Yaïche, R. Mazumdar and C. Rosenberg, “A game theoretic framework for bandwidth allocation and pricing in broadband networks,” IEEE/ACM Trans. Netw., vol. 8, no. 5, pp. 667-678, Oct. 2000.

[34] X.-R. Cao, H.-X. Shen, R. Milito and P. Wirth, “Internet pricing with a game theoretical approach: concepts and examples,” IEEE/ACM Trans. Netw., vol. 10, no. 2, pp. 208-216, April 2002.

[35] H. Park and M. van der Schaar, “Bargaining strategies for networked multimedia resource management,” IEEE Trans. Signal Processing, vol. 55, no. 7, pp. 3496-4622, July 2007.

[36] A. Rubinstein, “Perfect equilibrium in a bargaining model,” Econometrica 50(1):97-109, 1982.

[37] A. Rubinstein, “A bargaining model with incomplete information about time preferences,” Econometrica, vol. 53, no. 5, pp. 1151-1172, Sept. 1985.

[38] J. F. Nash, “Two-person cooperative games,” Econometrica 21:128-140, 1953.

[39] M. J. Osborne and A. Rubinstein, “A Course in Game Theory,” The MIT Press, Cambridge, Massachusetts, 1994.

[40] J. W. Pratt, “Risk Aversion in the Small and in the Large,” Econometrica, vol. 32, no. 1-2, pp. 122-136, Jan.-Apr., 1964.

[41] Andreas Berl, Andreas Fischer (editors), Autonomic Internet Project, Deliverable D1.1 – Initial Monitoring of Virtual Networks, 2009

[42] Jennings, B., Van Der Meer, S., Balasubramaniam, S., Botvich, D., Foghlu, MO, Donnelly, W. and Strassner, J., Towards autonomic management of communications networks, IEEE Communications Magazine, Volume 45, Number 10, pages 112-121, 2007.

[43] Strassner, J., Policy-Based Network Management: Solutions for the Next Generation (The Morgan Kaufmann Series in Networking), 2003, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

[44] S. Davy and B. Jennings and J. Strassner, The policy continuum-Policy authoring and conflict analysis, Elsevier Computer Communications, volume 31, number 13, 2008, issn 0140-3664, pages 2981-2995, Butterworth-Heinemann, Newton, MA, USA.

[45] Ponder2 policy framework, http://www.ponder2.net

[46] Maemo OS2007, http://maemo.org/downloads/OS2007

[47] Linux Kernel, http://en.wikipedia.org/wiki/Linux_kernel

[48] Debian, http://www.debian.org

[49] Maemo 3.x Bora SDK, http://maemo.org/development/sdks/maemo_3-x_bora

[50] Maemo 3.1 Bora SDK INSTALL.txt, http://tablets-dev.nokia.com/3.1/INSTALL.txt

[51] JamVM 1.4.3, http://jamvm.sourceforge.net

[52] Retrotranslator 1.2.9, http://retrotranslator.sourceforge.net

[53] GNU Classpath 0.91, http://www.gnu.org/software/classpath/downloads/downloads.html

[54] gconftool: http://projects.gnome.org/gconf

[55] dbus: http://packages.debian.org/search?searchon=sourcenames&keywords=dbus

[56] D. Woldegebreal, H. Karl, “Network-Coding-Based Cooperative Transmission in Wireless Sensor Networks: Diversity-Multiplexing Tradeoff and Coverage Area Extension,” EWSN 2008, LNCS 4913, pp. 141–155, 2008.

[57] M. Yu, H. Mokhtar, M. Merabti, "Self-Managed Fault Management in Wireless Sensor Networks," UBICOMM'08, pp. 13–18, 2008.

[58] S. Sheng, K. Li, W. Chan, Z. Xiangjun, D. Xianzhong, "Agent-Based Self-Healing Protection System," IEEE Transactions on Power Delivery, Vol. 21, No. 2, April 2006.

[59] Y. Wang, X. Li, Q. Zhang, "Efficient Self Protection Algorithms for static Wireless Sensor Networks," Global Telecommunications Conference, 2007. GLOBECOM '07, IEEE, 931-935, 2007

[60] N, Chohan, "Hardware Assisted Compression in Wireless Sensor Networks," http://www.cs.ucsb.edu/˜nchohan/docs/hadcwsnProgressReport.pdf, Aug. 2007.

[61] BOB hashing function: http://burtleburtle.net/bob/hash/doobs.html, Last accessed: September 2009.

[62] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, A. Lakhina, "Impact of Packet Sampling on Anomaly Detection Metrics," 6th ACM SIGCOMM Conference on Internet Measurements, Rio de Janeiro, Brazil, October 25-17, 2006, pp 159-164.

[63] G. van den Broek, J. Schoenwaelder, A. Pras, and M. Harvan "Snmp trace analysis definitions," In Resilient Networks and Services, Bremen, volume 5127 of Lecture Notes in Computer Science, pages 134–147, London, July 2008. Springer Verlag.

[64] Cisco White Paper, "Approaching the Zettabyte Era," June 2008.

[65] Cisco Application Extension Platform: http://www.cisco.com/en/US/products/ps9701/, Last accessed: September 2009.

[66] C. Estan, G. Varghese, "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice," ACM Transactions on Computer Systems, Volume 21, Issue 3, Aug. 2003.

[67] P. Maymounkov, D. Mazières: Kademlia, "A Peer-to-Peer Information System Based on the XOR Metric," IPTPS 2002, March 2002, Cambridge, MA, USA.

[68] J. Schoenwaelder, A. Pras, M. Harvan, J. Schippers, and R. van de Meent, "SNMP traffic analysis: Approaches, tools, and first results," In Proceedings of the Tenth International Symposium on Integrated Network Management, Munich, Germany, pages 324–332, Piscataway, May 2007. IEEE Computer Society Press.

[69] B. Trammell, E. Boschi, "Bidirectional Flow Export Using IP Flow Information Export (IPFIX)," RFC 5103, January 2008.

[70] T. Zseby, M. Molina, N. Duffield, S. Niccolini, F. Raspall, "Sampling and Filtering Techniques for IP Packet Selection," IETF RFC 5475, March 2009.

[71] Sopcast. http://www.sopcast.org, last visited: 26.01.2009.

[72] TomP2P, a distributed multi map, December 2008: http://www.csg.uzh.ch/publications/software/TomP2P.

[73] E. Adar and B. A. Huberman, "Free riding on gnutella. First Monday, 5:2000," 2000.

[74] S. Agarwal and S. Dube, "Gossip based streaming with incentives for peer collaboration," In Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on, pages 629–636, Dec. 2006.

[75] R. M. Axelrod, “The Evolution of Cooperation,” Basic Books, 1984.

[76] T. Bocek, Y. El-khatib, F. V. Hecht, D. Hausheer, and B. Stiller, “CompactPSH: An Efficient Transitive TFT Incentive Scheme for Peer-to-Peer Networks,” In 34th IEEE Conference on Local Computer Networks (LCN 2009), Zurich, Switzerland, October 2009.

[77] T. Bocek, W. Kun, F. V. Hecht, D. Hausheer, and B. Stiller, “PSH: A Private and Shared History-based Incentive Mechanism,” In 2nd International Conference on Autonomous Infrastructure, Management and Security Resilient Networks and Services (AIMS 2008), Bremen, Germany, July 2008.

[78] Y.-h. Chu, J. Chuang, and H. Zhang, “A case for taxation in peer-to-peer streaming broadcast,” In PINS ’04: Proceedings of the ACM SIGCOMM workshop on Practice and theory of incentives in networked systems, pages 205–212, New York, NY, USA, 2004. ACM.

[79] B. Cohen, “Incentives Build Robustness in BitTorrent,” In 1st Workshop on Economics of Peer-to-Peer Systems (P2PECON), Berkeley, CA, USA, June 2003.

[80] M. Feldman, K. Lai, I. Stoica, and J. Chuang, “Robust Incentive Techniques for Peer-to-Peer Networks,” In EC ’04: Proceedings of the 5th ACM conference on Electronic commerce, pages 102–111, New York, NY, USA, 2004. ACM Press.

[81] F. V. Hecht, T. Bocek, C. Morariu, D. Hausheer, and B. Stiller, “Liveshift: Peer-to-peer live streaming with distributed time-shifting,” Peer-to-Peer Computing, IEEE International Conference on, pages 187–188, 2008.

[82] Y. Huang, T. Z. Fu, D.-M. Chiu, J. C. Lui, and C. Huang, “Challenges, design and analysis of a large-scale p2p-vod system,” SIGCOMM Comput. Commun. Rev., 38(4):375–388, 2008.

[83] P. Maymounkov and D. Mazieres. Kademlia, “A peer-to-peer information system based on the xor metric,” in Proceedings of 1st International Workshop on Peer-to-Peer Systems (IPTPS), March 2002.

[84] J. J. D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. J. Epema, and H. J. Sips, “Give-to-Get: free-riding resilient video-on-demand in P2P systems,” In Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, volume 6818, Jan. 2008.

[85] V. Pai and A. E. Mohr, “Improving robustness of peer-to-peer streaming with incentives,” In Proceedings for NetEcon 2006, pages 31–36, Dec. 2006.

[86] A. Vlavianos, M. Iliofotou, and M. Faloutsos. Bitos, “Enhancing bittorrent for supporting streaming applications,” In INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings, pages 1–6, April 2006.

[87] Xiaojun Hei; Chao Liang; Jian Liang; Yong Liu; Ross, K.W., "A Measurement Study of a Large-Scale P2P IPTV System," *Multimedia, IEEE Transactions on* , vol.9, no.8, pp.1672-1687, Dec. 2007

[88] R. Landa, D. Griffin, R. Clegg, E. Mykoniati, and M. Rio, “A sybilproof indirect reciprocity mechanism for peer-to-peer networks,” In Proceedings of IEEE Infocom '09, 2009.

10 Abbreviations

AMS	Autonomic Management System
AS	Autonomous System
BSN	Body Sensor Network
CBR	Case Based Reasoning
CCR	Central Configuration Repository
DCBR	Database Case Based Reasoning
DHT	Distributed Hash Table
DiffServ	Differentiated Services
DOC	Distributed Orchestration Component
DP	Dynamic Planner
DRAM	Dynamic Random Access Memory
DSL	Domain-Specific Language
ECA	Event-Condition-Action
FI	Future Internet
FIN	Future Internet Network
G2G	Give-to-Get
HTN	Hierarchical Task Network
IDS	Intrusion Detection Systems
IPFIX	IP Flow Information Export
MANET	Mobile Ad-hoc Network
MO	Managed Object
MT	Multi-Topology
MTTR	Mean Time To Repair
ND	Network Dimensioning
NSA	Neighborhood Search Algorithm
OD	Orchestrated Domain
OP	Orchestration Plane
OS	Operating System
P2P	Peer-to-Peer
PBM	Policy Based Management
PC	Policy Consumer
PHB	Per-Hop Behavior

PMT	Policy Management Tool
PR	Policy Repository
QoE	Quality of Experience
QoS	Quality of Service
RID	Routing Identifier
SCRIPT	Scalable Real-Time IP Flow Record Analysis
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
SPE	Subgame Perfect Equilibrium
TFT	Tit-For-Tat
VoD	Video on Demand
WSN	Wireless Sensor Network

11 Acknowledgements

This deliverable was made possible due to the large and open help of the WP9 team of the EMANICS consortium within the Network of Excellence, which includes all of the deliverable authors as indicated in the document control. Many thanks to all of them.